A SPELLING ERROR DETECTION ALGORITHM FOR THE PDP11/03

by

RICHARD MICHAEL HELLER

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1981

Signature of
Author.................................................................
Deptment of Electrical Engineering, 5/1/81

Certified
by..............................................................
Thesis Supervisor

Accepted
by..............................................................
Chairman, Departmental Committee on Theses

ARCHIVE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 25 1981
LIBRARIES
A SPELLING ERROR DETECTION ALGORITHM FOR THE PDP11/03

RICHARD MICHAEL HELLER

Abstract

This thesis evaluates the advantages and disadvantages of two different approaches to detecting spelling errors in a text file, given the constraints of the PDP11/03 system. A batch algorithm is implemented and performance measures made. Program and data structure is discussed, and possible extensions noted.
# TABLE OF CONTENTS

| Introduction                                   | 1 |
| Software and hardware constraints             | 3 |
| Processing time--interactive approach         | 5 |
| Processing time--batch approach               | 6 |
| The dictionary                                | 8 |
| Minimization of memory usage                  | 10|
| Words                                         | 12|
| The batch algorithm                           | 13|
| Linked list implementation                    | 14|
| The checking function                         | 16|
| The correction function                       | 18|
| Editing the dictionary                        | 21|
| Measurements                                  | 23|
| Discussion                                    | 25|
| Recommendations for future work               | 27|
| Figures 1 - 6                                 | 28|
| **Appendix A** --GLOBAL CONSTANTS AND VARIABLES | 36|
| **Appendix B** --SPELL                        | 38|
| **Appendix C** --EDDCT                        | 61|
| **Appendix D** --GENDCT                       | 70|
| **Appendix E** --DLIST                        | 74|
INTRODUCTION

The problem of spelling error detection and correction in texts has been worked on for a number of years, and several useful systems have been built. (1) Peterson deals with many of the issues relating to error detection and correction. The program Peterson describes was an implementation using an interactive approach to detection; the detection proceeds one word at a time with incorrect words brought to the user's attention as they are found. The two main criteria for such a program are accuracy and speed. For a small computer of the size of the PDP11/03, the need for accuracy and speed suggest that the use of the interactive approach may not be ideal. A batch algorithm was selected to fit the constraints of the PDP11 system.

Several methods of error detection have been proposed based on calculation of the probability of certain letter combinations. These methods are, of course, language specific, and never completely accurate. In order to prevent any erroneous words from being missed by the spelling checker, a lookup method can be used based on a program dictionary. If the word in the text appears in the program dictionary, it is correct; if not, it should be brought to the attention of the user. Of course, even this

(1) Ralph Gorin, Spelling check/correction program on DEC-10, 1971 see also James L. Peterson, "Computer Programs for Detecting and Correcting Spelling Errors" in Communications of the ACM Dec. 1980
method is not foolproof, for the typographical error may actually take the form of another correct word (e.g., many for any). Context checking is another level of complexity entirely.

Because all acceptable words must appear in the dictionary, the dictionary is of necessity large. How this dictionary is and can be handled then becomes one of the major issues of the design effort. Since English has a large number of words (the Webster’s New International Dictionary, Second Edition contains 550,000 entries), it may appear impossible to deal with such a large number of words. Fortunately, all words do not occur with the same frequency. Some words are so rare that they need not be stored in the dictionary; if they do in fact appear in the text, chances are considerable that they are a typo for a more common word—in any case, they should be brought to the user’s attention. According to the word count in the American Heritage Word Frequency Book, the 50,000 most common words represent 99% of all words appearing within the literature surveyed. 10,000 words comprised 94%, 1000 words 74%, and 100 words 49%. Furthermore, certain words are used more often in certain contexts, so that in scientific texts, 10,000 words represent 98% of all words appearing. This suggests that a more limited sized dictionary would be suitable for many purposes.
SOFTWARE AND HARDWARE CONSTRAINTS

The language of program development was Pascal. Pascal is a strongly typed, block structured language, making for ease of coding and clarity of expression. A major objective of the design effort was to make the program easily understandable and maintainable for those who might wish to adapt and extend it in the future. A major liability of Pascal is its poor string manipulation functions. Foremost among these are the fact that string length must be fixed at compilation, and that string constants are not easily created.

The PDP11 has an address space of 32,768 words or 64KBytes. As Figure 1 shows, The RT11 vectors occupy the first 256 words, followed by the program code and global variables. Also resident is the RT11 monitor and the I/O page; these take 6 KW. Therefore, the program and memory have available a total of almost 26KW or 52KBytes. After the program and global variables are put in memory, the rest is available for use for local variables on the stack and dynamic memory from the heap. It is essential that the top of the heap not impinge upon the top of the stack, or a fatal run time error will occur. This factor is the major constraint on the program. At present, the program and global variables require about 10KW of memory. This leaves 16KW for the stack and heap. An interactive program might well prove to require less memory, but for the purposes of
evaluation we will say that it would use a comparable amount of memory.
PROCESSING TIME--INTERACTIVE APPROACH

Each text character is stored in one byte of memory. The length of strings was set at 16 characters. The Brown Corpus of words (see Peterson) shows 99% of all words to be 17 characters or less in length; 99% of all word appearances were of words 12 characters or less in length. Words in the text file longer than 16 characters would be in many cases concatenations of two or more words. With 16 bytes per word and 32K Bytes available, the maximum number of words which could be stored in memory is approximately 2000. Since an entire dictionary could not fit into memory, the dictionary as a whole must be kept on a secondary storage device (in this case a floppy disk). The most common format for an interactive program would be to have a cache of the most common words, and when a word is not in the cache, it would be checked for in a specific location in secondary storage. Practically, a cache would not exceed 1000 word, and it could therefore be expected from word distribution figures that approximately 25% of words would require disk accesses. The average access time for the disk drives which could be used are 483 msec for the RX01 and 320 msec for the CRDS disk drive. (2) If we the say it will take 1/2 second for every 4 words on the average, we compute a speed of .125 seconds per word. This extrapolates to about 4 minutes for

(2) PDP 11 Peripherals Handbook Charles River Data Systems Product Description
a 2000 word document. In actuality, the cache might be considerably smaller, as the larger the cache, the more time spent searching for words in them.

PROCESSING TIME--BATCH APPROACH

With the batch method, the words are read in all at once, and then compared to the dictionary in one process. Therefore, unlike the interactive method, where a specific portion of the dictionary may be read many times, in the batch approach the dictionary is read sequentially and only read once. An RX01 diskette may be read in 30 seconds during one sequential access. Though the entire dictionary is not read at once, much time can be saved as the drive head is not moved between reads. This leads to a maximum in the batch method, for 2000 words, of .015 seconds per word. In reality, the batch method requires considerable amount of overhead--the words must be placed in buckets, errors recorded, collected, and sorted. At present, the implemented algorithm has a limit of about 2000 words, though further passes through the input file and dictionary could extend this by 2000 words per pass (this has been partially implemented, but is not yet operational).

Spelling error correction provides another level of complexity. The most common typographical errors are: 1 missing letter, 1 extra letter, 1 incorrect letter, and transposed letters (Peterson). The magnitude of a correction
venture would be enormous. For instance, assuming the first letter of an incorrect word was wrong, there are 25 other letters which the word might start with. To check whether any of those words exist would basically require that the entire dictionary be read from disk. Since the user would have to approve the corrections in any case, the entire effort is left to the user.
THE DICTIONARY

The dictionary is a blocked file of characters. As in Figure 2, the block is comprised of a starting word, and end word and an array (called a page) of words. The size of the page has been selected to be 126 words. Together with the startword and the endword, this totals 128 16 character words and takes up 2048 bytes. The RX01 physical blocks are 512 bytes long, so this takes up 4 blocks.

The blocktable records the first and last words of each block, in an order corresponding to the order of blocks in the dictionary, along with a pointer for each block, which is used for storing text words (see below). It is stored in a file distinct from the dictionary file and is read in at the beginning of program execution. It is resident in memory at all times, as it is the crucial link between the dictionary and the program. The larger the number of blocks in the dictionary, the larger blocktable must be.

The dictionary and blocktable files are of constant size. The dictionary will not necessarily be full at all times. The user, using the EDDCT program can add or delete words from the dictionary. Several blocks at the end of the dictionary may be completely blank. Therefore, to further save time, the blocktable is read in only as far as the dictionary is full. When a entry in the blocktable is blank, there are no more non-empty blocks beyond that point in the dictionary. A blank set of entries are allocated for
mythical block 81, in case all 80 blocks are non-empty.

Non-empty dictionary blocks are completely full in most cases, but not always. The last non-empty block in the dictionary will in general not be full. Other blocks may not be full if a word has been deleted from the block. Startword will always be in wordentry 1, and all words in sequence until the endword will be valid. If there are any entry positions after endword, they do not contain valid entries. All blocks until the last non-empty blocks will contain at least one entry.

Dictionary size is fixed from generation and cannot be expanded by the EDDCT program, as changes are done by the random access approach, which precludes expansion. Random access allows blocks to be read from and written to the same file. Therefore, the entire file does not have to be written over, and editing can be done on a disk where the dictionary file takes up over half the storage capacity of the disk.

It is essential that the blocktable and the dictionary remain consistent. EDDCT does this by updating and rewriting the blocktable whenever the dictionary is altered. However, problems could occur with consistency if the user hits C or causes a fatal error between the times these tasks are accomplished. The operation is not atomic.
MINIMIZATION OF MEMORY USAGE

The major goals in the dictionary structure are minimizing the amount of memory used, which might imply a large number of small blocks read into memory one at a time, and minimizing time to read in the blocks, which might require larger blocks which can be read in fewer seeks. In actuality, there is an optimal size of the dictionary block in terms of minimizing memory usage, for as the number of blocks increase, the index table for blocks (blocktable) must become larger.

The equation for memory used by blocktable and 1 dictionary block is:

\[ \text{size(in bytes)} = \left( \frac{2 \times \text{stringlength}}{} \right) + 2 \times N + K/N \times \text{stringlength} + 32 \]

where \( N \) is the number of blocks into which the dictionary is divided and \( K \) the number of words in the dictionary. The first term is the size of the blocktable (including 2 bytes for a pointer), the second for one page of dictionary words, and 32 for the first and last words of a block. Differentiating and setting to zero to find a minimum,

\[ 0 = d\text{size}/dN = \left( \frac{2 \times \text{stringlength}}{} \right) + 2 \]

\[ + K \times \text{stringlength} \times (-1/N) \]

so

\[ \frac{2}{1/N} = \left( \frac{2 \times \text{stringlength}}{} + 2 \right) / \left( K \times \text{stringlength} \right) \]

\[ N = \text{sort}\left[ \left( K \times \text{stringlength} \right) / \left( \left( 2 \times \text{stringlength} \right) + 2 \right) \right] \]
If stringlength = 16 and K = 10,000 then N = 68 blocks and
\[ \text{pagesize} = \frac{10,000}{68} = 147 \text{ words.} \]
The closest practical pagesize to this is 126 words (adding 2 for
startword and endword, bringing the blocksize to 128 words). With 80
blocks, this leads to a dictionary of 10,080 words.

80 blocks does not seem excessive from the point of
view of speed. A lesser number of blocks might save time on
seeks, but might lose time elsewhere. With relatively small
blocks, it may occur that no words which would appear in a
particular block actually are in the input text. The
program, in this case, does not read in the block, but skips
to the next dictionary block.
Words are comprised of continuous sequences of alphabetic characters. Words are separated by all other characters. The apostrophe has been considered an alphabetic character if it is immediately preceded by an alphabetic character, as in "parent's" or "parents'". However, apostrophes starting words are not recognized (e.g., "'tis" is read as "tis"). Hyphenated words are broken up into their individual constituents, which are much more likely to appear in the dictionary.

If a word appears in the input file with the first letter capitalized, its case will be changed before looking it up in the dictionary, for in order for a match to occur, the characters in the word must have identical ASCII values. This means that normally capitalized words like IBM or MIT would appear in the dictionary as iBM and mIT. Succeeding letters in the word are not converted.
THE BATCH ALGORITHM

The batch method entails reading all words into memory before checking for correctness. Instances of incorrect words are then recorded, and the file passed through again, this time allowing corrections to be made. The file is read in until its end is reached, or until the program decides that there is not enough room in memory for more words.

As words are not checked in the order they appear in the text, some method must be used to keep track of where the words appear. One could simply record all incorrect words and then check for their appearance in the input file during the second pass. However, as these words would be unsorted, much processing time would be used doing comparisons. Instead, as a word is read in, it is associated with the index or count of its appearance in the file (e.g. "the" is associated with 4 if it is the fourth word in the file). Thus, only the count of the word need be recorded if an error is found. These can then be sorted and only a simple integer comparison done in the second pass.

When a word is read in, it is assigned to a bucket associated with the block in the dictionary within which, if it is a valid word, its match will be found. This is done by searching the blocktable until the block whose startword and endword frame the word in question, is found. For instance, if the word "grassle" appeared in the text, it
could only be correct if a matching entry is found in a block with, for instance, startword = friend and endword = gust.

When a word is added to a bucket, both the word and its count(index) are recorded. When a word appears again in the text, only the count is recorded. All counts of the same word are linked together. When a word is looked up, all occurrences of the word are immediately available, and all occurrences of mistakes can be recorded at the same time. This affords a significant speed advantage over the interactive approach, where all instances of words have to be checked separately.

LINKED LIST IMPLEMENTATION

This system of buckets is implemented through linked lists. As mentioned previously, each entry in the blocktable has associated with it a pointer. This pointer points to one of a series of records which each contain a word which should be looked up in that block. Each record(wordrecord) contains a string which is the word to be looked up, and two pointers. One pointer points to the next record on the chain, or nil if at the end, and the other points to another type of record(countrecord) which contains the integer count of the words appearance, and a pointer to other countrecords if the word appeared more than once. A typical bucket is shown in Figure 3. An exception to this
rule is when a word appears which would appear before the first word or after the last word of the dictionary (e.g., z200 after zoo). Such a word is clearly an error, but in order to allow it to be corrected through normal methods, they are assigned to the first block and last block respectively.

Linked lists are the appropriate choice here because they can grow dynamically and can be expanded easily. The memory size limits the amount of words which can be in memory, so conceivably words could be stored in fixed arrays with as much memory utilization as with linked lists. However, the distribution of words can not be anticipated beforehand, so the way the program associates words with blocks must be dynamic. In some files, words which belong in the first block might appear 50 times, while words which would appear in the second block only 5 times. Fixed array storage would lead to enormous wastage if the enough space for the maximum expected incidence had to be allotted to all blocks. Furthermore, the number of word instances the program can handle before exhausting memory is not fixed. If in one file, the same words are used several times, the actual words need only be stored once, and more word appearances can be handled than a file in which most words are used only one time. Once again, fixed arrays cannot efficiently cope with the vagarities of word distribution.
THE CHECKING FUNCTION

Once the text words have been stored in memory, the program is ready to check for correctness. This is done simply by reading in each dictionary block in sequence, and attempting to match each of the words in the bucket with the entries in the block. If no words are stored in a bucket, that dictionary block is skipped. If a word is found which is not in the dictionary, all the instances of the word are recorded by writing the counts in an errorfile. This errorfile is later read back in and sorted so the words can be found on the second pass with ease. A nicer implementation would be to keep the errors in memory. However, in order not to run out of space, correct wordrecords would have to be dynamically deallocated. This involves some complexity, but should be done to clean up the program.

During the checking operation, the memory is strained more than at any other time in the program. Besides all the text words in memory on the heap, a full page of the dictionary is stored in a local variable on the stack. Whenever this page is passed to subprocedures inside the checkblock procedure, it is passed as a 'var' variable, even though the page is not altered in any way. This var assignment is essential, as it prevents the page from being copied over several times in memory. When words are read in originally, enough space for a dictionary page is assured by
checking the space procedure to see if the stack pointer is a sufficient distance (minfree) away from $KORE$, the pointer to the top of the heap. The values of SP and $KORE$ are recovered through an embedded assembly language procedure. Once all words are checked, the linked lists in the buckets are deallocated using 'dispose.'
THE CORRECTION FUNCTION

On the second pass, the user is given the opportunity to correct those words flagged as errors by the program. Those parts which contain correct words cannot be edited. The mode of correction is as follows: the program recopies the input file until an error is reached. The user is then shown the erroneous spelling and has several options in ways to deal with the error. The user inputs the command, the program implements it, and then continues to copy over text until the next error is found. Since apparently erroneous spelling may appear more than once in the text, as well as words which are correct but which do not appear in the dictionary, the user can specify that some commands be executed on all occurrences of a word. As the user may wish to see the line on which the incorrect word appears, the input file is read in one line at a time on the second pass. Words are then recovered from this line (called currentline) and transferred to a temporary line (temp-line) until currentline is completely processed, when temp-line is then written to the output file. Lines are recorded with a maximum length of linemax (120) characters. As the program’s first pass on the input file reads words in one word at a time, there is the possibility of an inconsistency if an input line has more than 120 characters. Thus, the maximum line length is restricted to 120 characters (this should be changed in future versions).
The commands are:

A: ACCEPT WORD THIS ONCE
AF: ACCEPT FOREVER
AD: ACCEPT AND WRITE TO ADDFILE (TO BE ADDED TO THE DICTIONARY THROUGH EDDICT)
R: REPLACE WORD ONCE
RF: REPLACE FOREVER
D: DELETE WORD
L: DISPLAY CURRENT LINE OF TEXT
Q: QUIT (MAKE CHANGES UP TILL HERE AND ACCEPT REST OF FILE)
H: HELP (DISPLAYS LIST OF COMMANDS)

After a help command, or any illegal command, the user is given another opportunity with the same word. The H command does not alter the state of the program in any way.

When a command which is to be executed on a word more than once, like 'AF' is inputted, it is stored for further execution. The command is stored in record(cmdrecord), which contains the command (either 'A' or 'R'), the word which it refers to, and the replacement word, if there is one. These command records are checked before a word is brought to the attention of the user. Command records are stored as components of a linked list, as their number cannot be easily anticipated. In order to economize on memory space and search time, command records are stored only when the word is known to occur more than once.
The program knows whether an error has appeared more than once, as when errors are detected, all instances of the same word are recorded in the errorfile at the same time. Different words are delimited by 0's in the errorfile, as shown in Figure 4. The end of the file is signaled by the integer -1, as the OMSI Pascal eof function only works accurately with test files. When the errorfile is read in, the integers are encoded to show whether they are instances of a word which occurs more than once. The encoding equations are simply

\[ I = 2 \times \text{count} \quad \text{for single occurrences} \]

\[ I = (2 \times \text{count}) + 1 \quad \text{for multiple occurrences} \]

This encoding scheme maintains the same order of earliest appearance for sorting purposes, and can easily be decoded using integer division.
EDITING THE DICTIONARY

Utilities are provided to manage the dictionary. GENICT allows the user to generate a new dictionary from a list of words. The format of the list is one word per line. Only those words in sorted order are put in the dictionary. The dictionary is created at its mature size. All blocks after the last filled block are blank. A blocktable is also generated.

Addwords allows the dictionary's word base to be expanded. Since the dictionary may take up more than half the amount of storage on a disk, the dictionary is rewritten using the random access method. First, the addfile, which is in the same format as the generating file, but need not be sorted, is sorted. Then the block in which the first insert goes is read into memory(dictblock). It is copied over into a temporary block, insertions being made from the addfile where appropriate. This block is then written into memory. Some of the previous dictionary block is overflow, and must be stored in the next dictionary block, along with the rest of the next dictionary block. This next block will then have overflow, which ripples down until the end of the dictionary, where the overflow finds a home. This is illustrated in Figures 5a and 5b.

One danger of the random access method is that the overflow may become larger than one block. In normal circumstances, tempblock is written into the block just
stored in memory. However, if the overflow were larger than
one block, tempblock would be placed into a block which had
not yet been read, resulting in the loss of those words.
Therefore, additions have been restricted to one memory page
at a time.

Deletions are made from a file of format identical to
addfile. When a word is deleted, the block in which it was
resident is contracted, with a hole left at the end after
the endword. Because no blank blocks are allowed in the
middle of the dictionary, a word which is the only one in a
block cannot be deleted. The program will notify the user
or such an occurrence. In that unusual case, it is suggested
that a word or words be added to the dictionary before this
word, so that the part of the dictionary which contains this
block is recopied and compressed.

The dictionary may only be edited through the EDDCT
program. Addfiles are generated by the SPELL program, but
must be run through EDDCT for the dictionary to be altered.
This allows a dictionary manager to control all changes to
the dictionary.
MEASUREMENTS

A test run was made of 725 words of text (see Figure 6) taken from Chapter 8 of "The American Business Creed". (3) This is a fairly literate text employing a wider variety of words than might be expected in a scientific or computer oriented text. The dictionary was not complete but contained a list of 650 common words taken from the Thorndike and Lorge word count. (4) The dictionary contained 5 full blocks. For the 725 words, time measurements (including time to write out each word on the CRT) were:

To read in and store in buckets 35 sec.
To check correctness 25 sec.
To read in errorfile 20 sec.
To sort errorfile 5-10 sec.

Total time including time needed to reset and rewrite files comes to 2 minutes or about 6 words per second. If the list of errors were not written, but kept in memory, this could be cut down substantially. This figure is also more than would be expected if the dictionary were complete, as most of the words in the errorfile were actually correct but not

(4) Edward Thorndike, James Lorge, Teachers Book of Thirty Thousand Words
known words. When the RT11 command SET USR NOSWAP was used, processing time was cut to just 35 seconds, or 20 words per second. This command does limit somewhat the amount of memory available, but the increase in speed outweighs the loss in space.

Of the words processed, there were 204 distinct words which were not in the dictionary, and 87 repeat instances of these words. There were 23 actual typographical errors. At least 291/725 or 40% of the words would not have appeared in a cache. The time for these 291 disk accesses themselves would have been (for the CROS disk drive) 291 * 320 ms = 93 seconds. Time to process words would have resulted in substantially longer time.
The batch method appears to detect errors somewhat more swiftly than the interactive approach. However, the amount of time saved is perhaps less relevant than the distribution of processing time while the user is waiting for program response. For a file with a large number of errors, the interactive approach may be more convenient to the user. Every few seconds, the user would be presented with another error to be corrected. He would correct it, and wait a bit for the next word. A response time of a few seconds might be adequate. However, if a file contained only sporadic errors, the user might wait with his attention on the screen for long and irregular periods of time. The first error might appear in one second, the second five seconds later, and the next 30 seconds later. The user would not know how long he would have to wait before another word came up, but would have to wait patiently for the program. With the batch method, the user would have to wait initially for a long period of time, but once the program was on its correction pass, the user would be assured that errors would be outputted by the program one after another, swiftly and regularly. During the initial processing period, users could let the program churn away while they save their attention to some other matter. The main advantage for the batch approach seems, therefore, not to be in possible savings in calculation time, but in more convenient user
Engineering,

The batch method is promising for use on systems which do not have the ability to access parts of files at a random access basis. The program does read the dictionary blocks using OMSI Pascal’s random access feature so that blocks which are not needed may be skipped. However, since the dictionary is read only once, from beginning to end, it could just as easily be implemented by reading the dictionary through sequential access. EDDCT uses the random access feature because the same file can be read from and written to using it. However, if we restricted the dictionary size to less than half of the storage capacity of the system, a new copy of the dictionary could be written while the old copy is read, using standard sequential access methods. All disk accesses using the interactive approach would have to be done on a random access basis, as words are looked up in an order determined by the text, not the program.
RECOMMENDATIONS FOR FUTURE WORK

The addwords routine could be altered to allow overflows of several pages of words. Routines which allow the program to access and use the user's auxiliary dictionaries would be very useful.

Portions of the program could be overlaid, as most subsections of the program are used one at a time and in sequence. Much memory space could be saved with little loss of time.

The program could be adapted to remove common suffixes. If the correctness of the suffixes are to be checked (e.g., boxes is incorrect) there would have to be a link between a word and its allowable suffixes. To do this, a wholesale revision of the dictionary structure would be necessary. Suffix removal would allow more words to be stored in the dictionary, but would substantially increase processing time.
A PDP-11 program has an address space of 32,768 words, or 32KW (1KW is 1024 words). The figure below shows this address space as it might be allocated for a typical program of moderate size.

Figure 1
<table>
<thead>
<tr>
<th>startword</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>endword</td>
<td></td>
</tr>
</tbody>
</table>

| word 1   |   |
| word 2   |   |
| ...      |   |
| word 126 |   |

**Dictionary Block**
(Above)

**Blocktable**
(Below)

<table>
<thead>
<tr>
<th>startword</th>
<th></th>
<th>Block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>endword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bucket</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>startword</th>
<th></th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>endword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bucket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Block 81
Figure 3

Typical Bucket
| 119 | 0 | 64 | 0 | 16 | 29 | 83 | 0 | -1 |

Figure 4

Typical Errorfile
Figure 5a
Figure 5b
Generalized explanations of the over-all functioning of the economic system are naturally less frequent in the business creed than pronouncements on particular issues, such as the role of trade unions or the proper size of profits. While hostile ideologies attack the "system" in general terms, detailed discussion tends to focus on specific 'vices' of the system, and the defense of the system by business ideology must be similarly specific. In the bulk of business ideology, therefore, the general laws of operation of the system are assumed to be known; they are referred to symbolically, not explained in detail. These laws, formal and abstract in character, make poor material for advertisements, speeches, Congressional testimony, and pamphlets. Their exposition is for the most part confined to a few books--some of them textbooks--which deal at length with the whole system of major parts of it. (1) While these treatises share with other ideological media the usual characteristics of ideology--selectivity and symbolic appeal--they are necessarily constrained more by the demands of logical coherence and consistency.

The statement of general laws that govern the functioning of the whole economic system is, one might expect, a complicated task. Not so int the ideology: the economy is governed by natural, commonsense rules which anyone can understand--hence titles such as 'Shirtsleeve Economics' and 'Economics in One Lesson.' The NAM authors draw a sharp distinction between the concepts of academic economists 'so refined and so technical that (they) do not fit the realities of the business world' and their own simple analysis. The emphasis on simplicity and practicality is partly dialectical--abstraction and complexity are attributed to New Deal Professors and theorists. But it is also part of a basic anti-intellectual slant in the business ideology; 'practical knowledge' gained in meeting payrolls is far more valuable than speculative abstract thought.

THE CLASSICAL EXPLANATION: 'FUNCTIONAL' ARGUMENTS

The classical and managerial versions of how the economy functions are best exemplified in the NAM volumes and USA: The Permanent Revolution respectively. The classical explanation is like the model of a competitive private enterprise economy found in economics textbooks. It combines two quite different streams of economic thought. The first is the 'classical' economics of which the Physiocrats, Adam Smith and J. S. Mill were representative; the second is the 'neo-classical' writings
of Walras and Marshall. On what can be called the incentive level—the forces moving individuals in the market—the ideology borrows liberally from the laissez-faire views of the classical economists, and also from the view of the mature of human motivation held by early nineteenth-century utilitarians such as Jeremy Bentham. On what may be called the functional level of analysis—the prices, quantities, incomes, etc., generated by the system without much reference to the economic actors as individuals—the ideology selects from, and greatly dilutes, the neo-classical theory of perfectly competitive general equilibrium. These two aspects of the analysis are on somewhat different levels of abstraction. Analysis on the level of incentives is usually employed in the more popular explanations, such as Utley’s or Quenny’s, while the purely functional analysis is worked out in detail only in the relatively academic treatise of the NAM.

In comparison with what is offered in the classical version of the creed, the managerial explanation for the overall working of the competitive system is scanty. On the incentive level, the managerial version differs by placing emphasis on the responsible decisions of socially minded businessmen, absent in the classical version. On the functional level of analysis, the managerial creed simply lacks any fully defined explanation to supplant the competitive model of the classical creed. The managerial creed, in dealing with the individual enterprise, emphasizes the power of management decisions and the importance of responsible exercise of that power. However, when the creed deals with the whole economy, no similar single decision-making authority is envisaged; of course, yet there is no explanation—other than the competitive one of the classical creed—of just how the responsible decisions in each individual enterprise are coordinated for the whole economy. The result is a gap in the logical structure of the managerial version of the creed, which is usually filled with general reference to the forces of competition, but which lacks the detailed working-out of the classical creed.
APPENDIX A

GLOBAL CONSTANTS AND VARIABLES
const

stringmax=16; /*STRING LENGTH*/
linemax=120;
intmax=500;   /*LIMIT OF INTEGER ARRAY*/
errormax=intmax;
numberofblocks=80;
tabsize=81;    /*NUMBEROFBLOCKS + 1*/
two=2;
one=1;
zero=0;
neg1=-1;
apostrophe='''';
blank=' ';
padsize=126;
capoffset=32; /*difference between ascii upper and lower case*/
spfreq=50;    /*check memory available every 50 words*/
minfree=5000; /*5000 bytes left after word read in so there
               is space for a dictblock and more*/

type

blockrange=1..numberofblocks;
stringsrange=0..stringmax;
wordpointer~wordrecord;
countpointer~countrecord;
chararray=array[1..stringmax] of char;
intarray=array[1..intmax] of integer;
charline=array[1..linemax] of char;
tinfile=file of integer;
page=array[1..padsize] of chararray;

string= RECORD
          len : stringsrange;
          ch   : chararray;
end;

line= RECORD

len   : integer;
ch    : charline;
end;

entryInBlockTable=

RECORD
startword : chararray; /* DESCRIBES DICTIONARY BLOCK */
endword : chararray;

list : wordpointer; /* HOLDS BUCKET */
end;

wordrecord=
RECORD

word : chararray;
len : stringrange;

nextword : wordpointer;
wordplace : countpointer
end;

countrecord=
RECORD

wordcount : integer; /* SHOWS WHERE WORD APPEARS IN TEXT */
nextcount : countpointer
end;

block=
RECORD

startword : chararray;
endword : chararray;

wordentry : false
end;

blockfile=file of block;

var
errorfile : intfile;

inchar : char;

infilename, outfilename, addfilename, delname,
dictname, errorfilename, tablename : string;

blocktable : array[1..tabsize] of entryInBlockTable;

infile : text;
infilecount : integer; /* FOR MULTIPLE PASSES THRU READINPUTFILE, ONLY PARITALLY IMPLEMENTED */

emptyword : chararray;
APPENDIX B

CONSTITUENT PROGRAMS OF SPELL.SAV
SPELL

Major Procedures

- CLEAR
- READSTRING
- READFILE
- INITBLOCKTABLE
- INITREADIN
- READINPUTFILE
  - SPACE
  - ADDTOTREE
    - ADDWORDS
    - ADDCOUNT
    - READWORD
      - BUILDWORD
- CHECKFILE
  - INITDICT
    - READBLOCK
  - CHECKBLOCK
    - WORDOKAY
    - WRITEERROR
    - UPDATEBLOCKS
      - READBLOCK
- KILLLISTS
- CORRECTFILE

(next page)
function space : boolean;

var
diff, stackP, topheap : INTEGER;

begin

(*$C .GLOBL $KORE
MOV $KORE, TOPHEAP(SP) *)
(*$C MOV SP, STACKP(SP)
*)

/* kore points to top of space allocated by new */

writeln;
writeln(  'kore ',topheap);
writeln(  'SP ', stackP);
if ( (stackP <= 0) and (topheap <= 0) )
or ( (stackP >= 0) and (topheap >= 0) )
then
begin
  diff:=abs(stackP - topheap); /*sp usually negative( >32k )* /
  if (diff > minfree) then
    space:=true
  else space:=false
end
else
begin
  stackP:=stackP + maxint; /*measuring from middle of address
space (which goes from maxint to
-1 as number is considered
2's complement */
  topheap:=maxint - topheap;
  if (stackP > minfree) or (topheap > minfree) then
    space:=true
  else space:=false /*minimum that must be left for checkfile's
local variables*/
end
```plaintext
procedure clear(var s : chararray);
  var i: integer;
  begin
    for i:=1 to stringmax do s[i]:=blank
  end; /* clear */

procedure readstring(var s: string); /* works only for tty input */
  begin
    clear(s, ch);
    s.len:=0;
    with s do
      while (not eoln) and (len < stringmax) do
        begin
          len:=len + 1;
          read(ch[len])
        end;
    readln
  end; /* readstring */

procedure readfile(f: text; var s: string); /* works only for file input */
  begin
    clear(s, ch);
    s.len:=0;
    with s do
      while (not eoln(f)) and (len < stringmax) do
        begin
          len:=len + 1;
          read(f, ch[len])
        end;
    readln(f)
  end; /* readfile */

procedure initblocktable(tablename: string);

  type
    cfile= file of chararray;

  var
    tablefile : cfile;
    i : integer;
    buffer : chararray;

  begin
    reset(tablefile, tablename.ch, 'tab');
    i:=0;
    repeat
      /* read startword and endword until blank marker, no more non-empty blocks after this */
      begin
```
i:=i+1;
buffer:=tablefile~;
set(tablefile);
blocktable[i].startword:=buffer;
buffer:=tablefile~;
set(tablefile);
blocktable[i].endword:=buffer;
blocktable[i].list:=nil
end
until blocktable[i].startword=emptyword

end; /*initblocktable*/

/***************************************************************************/

procedure initreadin(filename : string);

begin
    reset(infile, filename.ch,'txt');
infilecount:=0       /*first pass, words start from base=0*/
end;

/***************************************************************************/

procedure addtotree(var word : string; var count : integer);

var
    index : integer;
    wordlink : wordpointer;

procedure addword(var word : string; var index, count : integer);

var
    wordp : wordpointer;
    countp : countpointer;

begin
    /* clear array of char */
    new(wordp);      /*new wordrecord, first instance of word*/
    new(countp);
    wordp^.word:=word.ch;
    wordp^.len:=word.len;
    wordp^.nextword:=blocktable[index].list; /*put into bucket*/
    blocktable[index].list:=wordp;
    wordp^.wordplace:=countp;
    countp^.wordcount:=count + infilecount; /*if on another pass
    thru dictionary, the count starts from a base of the lastword readin
    on previous pass */
    countp^.nextcount:=nil
end; /*addword*/

procedure addcount(var wordlink : wordpointer; var count : integer);
var
countp: countpointer;

begin
  new(countp);
  countp^.nextcount := wordlink^.wordplace;
  wordlink^.wordplace := countp;
  countp^.wordcount := count + infilcount
end; /*addcount*/

begin /*adddtotree adds to bucket */
  if word.ch <> emptyword then
    begin
      index := 0;
      repeat
        index := index + 1
      until ( (word.ch <= blocktable[index].endword) 
        or (blocktable[index].startword = emptyword) )
        /*find which block its in*/

      if blocktable[index].startword = emptyword then
        index := index - 1;
        /*its not in dict, but we have to set 
it in the tree, so writeerror sets it 
so we put it in last block */

      wordlink := blocktable[index].list;

      while (wordlink <> nil) and 
        (wordlink^.word <> word.ch) do
        wordlink := wordlink^.nextword; /*is word already in bucket*/

      if wordlink = nil then 
        addword(word, index, count) /*new word */
      else addcount(wordlink, count) /*word appeared before */
    end
  end;

/**************************************************************/

procedure readword(var fil: text; var word: string);

var
  charac: char;
  temp: integer;

procedure buildword(character: char; var word: string);

begin
  /*first letter in word can be capitalized, 
  middle letters can't. So if word has 
  capital in middle, it will still be read 
as one word but it will register an error 
when looked up */

  while ( (character >= 'a') and (character <= 'z') )
  or ( (character >= 'A') and (character <= 'Z') )
  or (character = apostrophe) do
begin
with word do
begin
    len:=len + 1;
    ch[len]:=character;
    if len < stringmax then
        read(fil, character) /* what of eof*/
    else character:=blank
end
end /*with*/ /*if string too long, we read it up to stringmax, then quit this cycle, rest of word read in as separate word */
end; /*buildword */

begin
    clear(word.ch);
    word.len:=0;
    if not eof(fil) then
        begin
            repeat
                begin
                    read(fil, charac);
                    if (charac >= 'A' and (charac <= 'Z')) then
                        begin
                            /* first letter can be capitalized*/
                            temp:=ord(charac) + capoffset;
                            charac:=chr(temp)
                        end
                    end
                until eof(fil) /* what happens to last letter*/
                or
                ( (charac >= 'a' and (charac <= 'z')) );
                if not eof(fil) then
                    buildword(charac,word) /* set rest of word */
            end
        end; /*readword*/
    end; /*buildword*/

procedure readInPutFile(var infile count : integer);

var
    word : string;
    count : integer;
    spaceleft : boolean;

function space : boolean; external;

begin
    count:=0;
    spaceleft:=true;
    while (not eof(infile))
        and spaceleft do
            begin
                count:=count + 1; /* read word, put in bucket*/
                readword(infile, word);
                addtotree(word, count);
                if (count mod spfreq)=0
then spaceleft:=space; /*check space in memory
   every space free words. if running out, quit reading in words */

   infilcount:=infilcount + count
end;

*************************************************************************

procedure killlists;

var
   i : integer;
   wordlink, p : wordpointer;
   countlink, a : countpointer;
begin
   /*now there will be ample memory*/
   for i:= 1 to numberofblocks do
      begin
         wordlink:=blocktable[i].list;
         while wordlink <> nil do
            begin
               countlink:=wordlink^.wordplace;
               p:=wordlink;
               wordlink:=wordlink^.nextword;
               dispose(p);
               while countlink <> nil do
                  begin
                     a:=countlink;
                     countlink:=countlink^.nextcount;
                     dispose(a)
                  end;
            blocktable[i].list:=nil
         end
      end
end;
Procedure clear(var s : chararray); external;

Procedure readstring(var s: string); /* works only for tty input */ external;

Procedure readblock(dictfile : blockfile;
    index : blockrange; var chosenblock : block);

begin
    seek(dictfile, index); /* random access */
    chosenblock:=dictfile^
end;

Procedure checkfile(dictname : string; errorfile : intfile);

var
    Presentblock : block;
    dictfile : blockfile;
    index : blockrange;

Procedure initdict(dictname : string; var Presentblock : block);

begin
    reset(dictfile, dictname.ch,'dct/seek');
    readblock(dictfile, one, Presentblock)
end;

Procedure checkblock(index : blockrange; var Presentblock : block);

    /* block is var so its not copied in memory */

var
    wordlink : wordpointer;

Procedure writeerror(err : wordrecord);

var
    countlink : countpointer;

begin
    countlink:=err.wordPlace;
    while countlink <> nil do
        begin
            errorfile^:=countlink^.wordcount;
            put(errorfile);
            countlink:=countlink^.nextcount
        end;
    errorfile^:=zero; /* delimits words by 0's */
    put(errorfile)
end;

function wordOkay(word : chararray;
    var Presentblock : block) : boolean;

var
  i : integer;
begin
  i:=1;
  while (presentblock.wordentry[i] <> word)
    and (presentblock.wordentry[i] <> presentblock.endword) do
    i:=i + 1;
  if presentblock.wordentry[i] <> word
    then wordokay:=false
    else wordokay:=true
end;

begin
  wordlink:=blocktable[index].list;  /*bucket*/
  while wordlink <> nil do
    begin
      if not wordokay(wordlink^.word, presentblock)
        then
          writeerror(wordlink^);
          wordlink:=wordlink^.nextword;
    end
end;

procedure updateblocks(index : blockrange; var presentblock : block);
begin
  if blocktable[index + 1].list <> nil then
    readblock(dictfile, index + 1, presentblock)
  else
    clear(presentblock, startword)  /*skip block*/
end;

begin
  initdict(dictname, presentblock);
  index:=1;
  while blocktable[index].startword <> emptyword do
    begin
      if blocktable[index].list <> nil then
        checkblock(index, presentblock);
        updateblocks(index, presentblock);
        index:=index + 1
    end;

  errorfile^:=nesi;  /*end of file */
  put(errorfile);
  close(errorfile)
end;
function single(n : integer) : integer;
begin
    single := 2 * n /* encodes that word occurs once(per pass)*/
end;

function multiple(n : integer) : integer;
begin
    multiple := (2 * n) + 1
end;

procedure readerrorfile(errorfname : string; var errorarray : intarray;
var size : integer);
var
    i, buffer, oldbuffer : integer;
    finished : boolean;
    errorfile : intfile;
begin
    writeln('reading errorfile...');
    reset(errorfile, errorfname, ch, 'err');
    i := 0;
    finished := false;

    while not finished do
        begin
        buffer := errorfile^;
        set(errorfile);
        if buffer < 0 then
            begin
            size := i; /*eof not accurate*/
            finished := true
            end
        else
            begin
            i := i + 1;
            oldbuffer := buffer;
            buffer := errorfile^; /*last check if two counts are together*/
            set(errorfile);

            if buffer = 0 then
                errorarray[i] := single(oldbuffer) /*word occurs once*/
            else
                begin
                repeat
                begin
                errorarray[i] := multiple(oldbuffer); /*appears > 1 times*/
                oldbuffer := buffer;
                end
                until buffer < 0
            end
        end
    end;
end;
buffer:=errorfile\^;
set(errorfile);
i:=i + 1
end
until (buffer=0) or (i >= errormax);

if i <= errormax then
    errorarray[i]:=multiple(oldbuffer)
else
    begin
        writeln('number of errors exceeds program capacity.');
        writeln('a second run will be needed.');
    end;
if i >= errormax then
    begin
        finished:=true;
        size:=errormax
    end
end /* if = 0 */
end /*if < 0 */
end /*not finished*/
end; /* readerrorfile */
procedure sortintegers(var errorarray : intarray; size : integer;
var
    minnum, base, k, mino : integer;
begin
    for base:=1 to size do
    begin
        minnum:=errorarray[base];
        mino:=base;
        k:=base;
        while k < size do
        begin
            k:=k + 1;
            if errorarray[k] < minnum then
            begin
                minnum:=errorarray[k];
                mino:=k
            end;
        end;
        errorarray[mino]:=errorarray[base];  /*exchange sort*/
        errorarray[base]:=minnum
    end;  /*sortintegers*/
Procedure clearline(var s: line);

var
  i : integer; /*clear whole line*/

begin
  for i:=1 to linemax do s.ch[i]:=blank;
  s.len:=0
end;

Procedure readline(fil : text; var s : line);
begin
  clearline(s);
  with s do
  begin
    while (not eoln(fil)) and (len < linemax) do
    begin
      len:=len + 1;
      read(fil,ch[len])
    end;
    readln(fil)
  end;
end;

/*text is read and written line by line. currentline is
the input line, templine the output line. nextwo passes thru
line to set word, setword copies over spaces and punctuation
between words, makeword retrieves words, cstart keeps track of
where you are in currentline, tstart on templine, cstart is not
changed in this module when a word is retrieved; that occurs in
exco.pas */

Procedure makeword(currentline : line; cstart : integer;
var word : strings);

var
  i, j : integer;

begin
  i:=1;
  j:=cstart; /*start of word on currentline*/
  repeat
    begin
      word.ch[i]:=currentline.ch[j];
      i:=i + 1;
      j:=j + 1
    end /*until you do past last letter in word*/
  until (not ( (currentline.ch[j] >= 'A') and
                (currentline.ch[j] <= 'Z') )
        or
                (currentline.ch[j] >= 'a') and
                (currentline.ch[j] <= 'z') )
end;
or (j > currentline.len) or (i > stringmax); /* splits into 2 words */
word.len=1 - i /* overshot */
end; /* make word */

Procedure setword(currentline : line;
var templine : line; var cstart, tstart : index;
var word : strings; outfile : text);

begin
i := integer;
copyover := boolean;

begin
 copyover:=true;
 while copyover do
 begin
 if (cstart > currentline.len) then copyover:=false;
 if (tstart > linemax) then copyover:=false;
 if (currentline.ch[cstart] >= 'A' and
 currentline.ch[cstart] <= 'Z') then
 copyover:=false;
 if (currentline.ch[cstart] >= 'a' and
 currentline.ch[cstart] <= 'z') then
 copyover:=false;

 if copyover then
 begin
 /* non alphabetical characters */
templine.ch[tstart]:=currentline.ch[cstart];
tstart:=tstart + 1;
cstart:=cstart + 1;
end; /* just copy over file */
end;
if (tstart > linemax)
and (cstart <= currentline.len) then
/* templine filled, currentline
still has stuff left */
begin
writeln(outfile, templine.ch : linemax); /* adds CR */

for i:=cstart to currentline.len do
 write(currentline.ch[i]);
writeln(' truncated from ');
writeln(templine.ch : linemax);
clearline(templine)
end;

if (cstart > currentline.len) then /* usual case, all of currentline has
been processed, content of line overwrite */
begin
templine.len:=tstart - 1; /* correct overshoot */
writeln(outfile, templine.ch : templine.len);
clearline(templine)
end;

if (tstart <= linemax)
and ( (currentline.ch[cstart] >= 'A') and
(currentline.ch[cstart] <= 'Z')
      or
      ( (currentline.ch[cstart] >= 'a') and
         (currentline.ch[cstart] <= 'z') )

    then
      makeword(currentline, cstart, word) /*1st letter can’t be upper*/
    end; /*setword*/

procedure nextword(infile, outfile : text; var currentline : line;
                   var tempLine : line; var cstart, tstart : int;
                   var word : string);

var
  finished : boolean;

begin
  finished:=false;
  word.ch:=emptyword;
  word.len:=0;
  while ( (word.len=0) and (not finished) ) do begin
    setword(currentline, tempLine, cstart,
            tstart, word, outfile);
    if word.len=0 then
      if (not eof(infile)) then /*no more words on line*/
        begin
          readline(infile, currentline); /*set next*/
          cstart:=1;
          tstart:=1
        end
      else
        finished:=true
    end; /*while */
  end; /*nextword*/

procedure Place(word : string;
                var tempLine : line; var tstart : integer);

var
  room, k : integer;

begin
  room:=(linemax - tstart) + 1;
  if word.len > room then
    begin
      for k:=1 to room do begin
        tempLine.ch[tstart]:=word.ch[k];
        tstart:=tstart + 1
      end;
      for k:= (room + 1) to word.len do
        write(word.ch[k]);
      write(' truncated ')
    end
  else
    for k:=1 to word.len do
      begin

templine.ch[tstart]:=word.ch[k];
tstart:=tstart + 1
end
end;

procedure accept(word : string; var templine : line; 
var cstart, tstart : integer);
begin
  cstart:=cstart + word.len; /*now at space after word*/
  place(word, templine, tstart) /*tstart changes*/
end;

procedure replace(oldword, newword : strings; var templine : line; 
var cstart, tstart : integer);
begin
  cstart:=cstart + oldword.len;
  place(newword, templine, tstart)
end;

procedure delete(word : string; currentline : line; 
var cstart : integer);
begin /* currentline here for readability */
  cstart:=cstart + word.len
end;
Procedure correctfile(infilename, outfilename, errorfilename,
    addfilename : strings);

    var
        errorarray : intarray;
        errmsg : integer;

    Procedure clearline(var s : line); external;
    Procedure clear(var s : chararray); external;
    Procedure readstrings(var s : strings); /*works only for its input*/ external;
    Procedure readline(fil : text; var s : line);
    external;
    Procedure readerrfile(errorfilename : strings; var errorarray : intarray;
        var size : integer);
    external;

    Procedure sortintegers(var errorarray : intarray; size: integer);
    external;

/***************************************************************************/

Procedure execCorrections(infilename, outfilename, addfilename : strings;
    var errorarray : intarray; errmsg : integer);

type
   .cmdpointer=^cmdrecord;
    cmdrecord=

    RECORD
        com : strings;
        oldword : strings;
        newword : strings;
        link : cmdpointer

    end; 

    var
        infile, outfile, addfile : text;
        errorindex, nexterr, count, cstart, tstart : integer;
        word, newword : strings;
        currentline, templine : line;
        cmdlink : cmdpointer; /*POINTS TO LIST OF CMDRECORDS*/
        user : cmdrecord;
        quit, recycle : boolean;
function demultiple(n : integer) : boolean;

begin
  if (n mod 2) = 1 then
    demultiple := true  /*DECODES WHETHER WORD APPEARS ONCE*/
  else demultiple := false
end;

procedure makeword(currentline : line; cstart : integer;
  var word : string);
  external;

procedure setword(currentline : line;
  var templine : line; var cstart, tstart : integer;
  var word : string; outfile : text);
  external;

procedure nextword(infile, outfile : text; var currentline : line;
  var templine : line; var cstart, tstart : integer;
  var word : string);
  external;

procedure addcom(var cmdlink : cmdpointer; user : cmdrecord);

var
  p : cmdpointer;

begin
  new(p);
  p^ := user;  /*ADD A CMDRECORD TO THE LIST*/
  p^.link := cmdlink;
  cmdlink := p
end;

procedure searchcom(word : string; cmdlink : cmdpointer;
  var user : cmdrecord);

var
  userlink : cmdpointer;

begin
  userlink := cmdlink;
  while (userlink <> nil) and
    (word.ch <> userlink^.oldword.ch) do
    userlink := userlink^.link;
  if userlink <> nil then
    /*COMMAND ALREADY REQUESTED ON WORD*/
    user! := userlink^;
end;

procedure place(word : string;
  var templine : line; var tstart : integer);
  external;

procedure accept(word : string; var templine : line;
  var cstart, tstart : integer);
  external;
procedure replace(oldword, newword : strings; var temp\line: line;
  var cstart, tstart : integer):
  external;

procedure delete(word : strings; currentline : line;
  var cstart : integer):
  external;

begin
  reset(infile, infilename.ch, 'txt');
  rewrite(outfile, outfilename.ch, 'txt');
  rewrite(addfile, addfilename.ch, 'add');

  cmdlink:=nil;
  count:=0;
  errorindex:=0;
  quit:=false;
  clear(word.ch);
  word.len:=0;
  getline(temp\line);
  readline(infile, currentline);

  tstart:=1;
  cstart:=1;

  while not quit do
    begin
      errorindex:=errorindex + 1;
      if errorindex <= errsize then
        nexterr:=errorarray[errorindex] div 2
        /* decodes entries */
      else nexterr:=maxint; /* so it keeps processing till end */

      repeat
        begin
          nextword(infile, outfile, currentline, temp\line,
            cstart, tstart, word);
          count:=count + 1;
          if word.len=0 then
            quit:=true /* END OF FILE */
          else
            if count <> nexterr then /* SPelled CORRECTLy */
              accept(word, temp\line, cstart, tstart)
        end
      until (count=nexterr) or quit;

      if not quit then
        begin
          recycle:=true; /* RECYCLE IF H OR INCORRECT COMMAND */
          while recycle=true do
            begin
              recycle:=false;
              clear(user.com.ch);
              clear(user.oldword.ch);
              clear(user.newword.ch);
              if demultiple(errorarray[errorindex]) then /* WORD APPEARS MORE
searchcom(word, cmdlink, user);

if user.com.ch[one]=blank then /* single or 1st of multiple */
  begin
    writeln(word.ch : word.len);
    write('?');
    readstrings(user.com); //ASK USER FOR COMMAND*/
  end;

case user.com.ch[one] of
  'a': case user.com.ch[two] of
    'f': begin
      if demultiple(errorarray[errorindex]) then
        begin
          user.com.ch[two]:=blank; /*DELETE F, SO NEW
          CMDRECORD NOT CREATED
          EVERY TIME WORD APPEARS*/
          user.oldword:=word;
          addcom(cmdlink, user);
        end;
      accept(word, templine, cstart, tstart)
    end;
    'd': begin
      if demultiple(errorarray[errorindex]) then
        begin
          user.com.ch[two]:=blank;
          user.oldword:=word;
          user.newword:=newword;
          addcom(cmdlink, user)
        end;
      writeln(addfile, word.ch); /*ADD TO DICT WITH
      EDDICT PROGRAM */
      accept(word, templine, cstart, tstart)
    end;
  blank: accept(word, templine, cstart, tstart);
  else begin
    writeln('incorrect command, type h for help.');
    recycle:=true
  end
end; /*CASE OF A */

'd': delete(word, currentline, cstart);

'r': case user.com.ch[two] of
  'f': begin
    write('newword? ');
    readstrings(newword); /* declare this */
    if demultiple(errorarray[errorindex]) then
      begin
        user.com.ch[two]:=blank;
        user.oldword:=word;
        user.newword:=newword;
        addcom(cmdlink, user)
end;
replace(word, newword, templine, cstart, tstart)
end;

blank: begin
if user.newword.ch[one]=blank then
begin
write('newword? ');  
readstring(newword);
user.newword:=newword
end;
replace(word, user.newword, templine,  
cstart, tstart)
end;
else begin
writeln('incorrect command, type h for help.');  
recycle:=true
end
end;  /*CASE OF R*/

'a': begin
  /*accept rest of file*/
accept(word, templine, cstart, tstart);
repeat
begin
nextword(infile, outfile, currentline, templine,  
cstart, tstart, word);
if word.len=0 then
  quit:=true
else
  accept(word, templine, cstart, tstart)
end
until quit; /*ACCEPT TO EOF AS IS */
end;  /* Q */

'l': begin
writeln(currentline.ch: currentline.len);
writeln(templine.ch: tstart);
recycle:=true
end;

'h': begin
writeln;
writeln('  COMMAND LIST  ');
writeln('a  accept');
writeln('af  accept forever');
writeln('ad  accept and store in addfile');
writeln('r  replace (wait for prompt)');
writeln('rf  replace forever');
writeln('d  delete');
writeln('l  display textline');
writeln('q  quit (accept rest of file)');
writeln('h  displays this message');
recycle:=true
end;

else begin
writeln('incorrect command, type h for help.');  
recycle:=true
end

end /*CASE USER.COM OF */
end /*while recycle*/
end /* if not quit */
end; /* while */
close(addfile);
close(outfile)
end;

begin

readerrfile(errorfname, errorarray, errsize);
sortintegers(errorarray, errsize);
execCorrections(infilname, outfilname, addfname,
                errorarray, errsize)
end; /* correctfile */
procedure clear(var s : chararray); external;
procedure readstrings(var s:string); /* works only for tty input */ external;
procedure initblocktable(tablename: strings); external;

procedure initreadin(filename : strings); external;
procedure readinputfile(var infilecount : intser); external;
procedure checkfile(dictname : strings; errorfile : intfile);
    external;
procedure killlists; external;

procedure correctfile(infilname, outfilname, errorfname, addfname : strings);
    external;

begin
    clear(emptyword);
    writeln('enter infilename');
    readstrings(infilname);
    writeln('enter outfilename');
    readstrings(outfilname);
    writeln('enter errorfilename');
    readstrings(errorfname);
    writeln('enter tablefilename');
    readstrings(tablename);
    writeln('enter dictionary name in form file,ext/seek');
    readstrings(dictname);
    writeln('enter addfilename');
    readstrings(addfname);
    initblocktable(tablename);
    initreadin(infilname);
    writeln('reading input file into memory...');
    readinputfile(infilecount);
    rewrite(errorfile, errorfname, ch, 'err');
    writeln('checking words...');
    checkfile(dictname, errorfile);
    killlists;
    correctfile(infilname, outfilname, errorfname, addfname)
APPENDIX C

CONSTITUENT PROGRAMS OF EIDCT.SAV
EDDCT

Major Procedures

- CLEAR
- READSTRING
- READFILE
- INITBLOCKTABLE
- ADDWORDS
  - SORTWORDS
  - INITADD
  - INSERTWORDS
    - READBLOCK
    - PUTBLOCK
    - WRITEBLOCKTABLE
- DELWORDS
  - SORTWORDS
  - CUTWORDS
    - READBLOCK
    - PUTBLOCK
    - WRITEBLOCKTABLE
procedure writeBlocktable;

type
cfile=file of chararray;

var
tablefile : cfile;
index : integer;
tablename : string;

procedure clear(var s : chararray);
var i; integer;
begin
  for i:=1 to stringmax do s[i]:=blank
end;/*clear*/

procedure readstrings(var s:strings); /*for tty input*/

begin
  clear(s,ch);
s.len:=0;
  with s do
    while (not eoln) and (len < stringmax) do
      begin
        len:=len + 1;
        read(ch[len])
      end;
  readln
end; /*readstrings*/

begin
  /*writes pairs of entries in blocktable as chararrays, not as entryInBlocktable, which includes pointer*/

  writeln('enter tablefile name');
  readstrings(tablename);
  rewrite(tablefile, tablename, ch, 'tab');
  for index:=1 to numberofblocks do
    begin
      tablefile^:=blocktable[index].startword;
      put(tablefile);
      tablefile^:=blocktable[index].endword;
      put(tablefile)
    end;
clear(blocktable[index].startword); /*blank marks end*/
tablefile^:=blocktable[index].startword;
put(tablefile);
tablefile^:=blocktable[index].startword;
put(tablefile);
close(tablefile)
end;
procedure sortwords(var addarray : pase; size : inteser);

var
    base, k, min : integer;
    minword : chararray; /*same as sortintesers*/

begin
    for base:=1 to size do
    begin
        minword:=addarray[base];
        min:=base;
        k:=base;
        while k < size do
        begin
            k:=k + 1;
            if addarray[k] < minword then
            begin
                minword:=addarray[k];
                min:=k;
                end;
            end;
        addarray[min]:=addarray[base]; /*exchanse sort */
        addarray[base]:=minword
    end;
    for base:=1 to size do write(addarray[base]: stringmax)
end; /*sortwords*/

procedure writenblock(dictfile : blockfile; dictblock : block; k : blockrange

begin
    seek(dictfile, k);
    dictfile":=dictblock;
    put(dictfile);
end;

procedure addwords(addname, dictname : string);

var
    charac : char;
    dictfile: blockfile;
    addarray : pase; /*limits no. of adds*/
    addfile : text;
    buffer : string;
    temp, firstinsert, i : integer;

procedure clear(var s: chararray); external;
procedure writeblocktable;
procedure readline(fil: text; var s: string); /* for file input */

begin
  clear(s.ch);
  s.len:=0;
  with s do
    while (not eoln(fil)) and (len < stringmax) do begin
      len:=len + 1;
      read(fil,ch[len])
    end;
    readln(fil)
  end; /*readfile*/

procedure initadd(dictname: string;
                  firstword: chararray; var firstinsert: integer);

var
  index: integer;

begin
  reset(dictfile, dictname.ch,'dct/seek');
  index:=0;
  repeat
    index:=index + 1
    until blocktable[index].endword > firstword;
  firstinsert:=index; /*up till here, dict need not be changed*/
end; /*initadd*/

procedure insertwords(addarray: page; size, firstinsert: integer;
                        dictfile: blockfile);

var
  i, dictindex, tempindex, numoldblock, numnewblock: integer;
dictblock, tempblock: block;
finished: boolean;
newword: chararray;

procedure readblock(dictfile: blockfile;
                     index: blockrange; var chosenblock: block);
  external;

begin
  tempindex:=1;
dictindex:=1;
  numoldblock:=1;
  numnewblock:=1;
  readblock(dictfile, firstinsert, dictblock);
  finished:=false;
i:=1;
newword:=addarray[i]; /*nextword to be added*/
while not finished do begin
  if (dictblock.wordentry[dictindex] <= newword)
or (newword=emptyword)
then
begin
  if newword=dictblock.wordentry[dictindex] then
    begin
      i:=i + 1;
      if i <= size then
        newword:=addarray[i]
      else
        clear(newword)
    end
  else
begin
  tempblock.wordentry[tempindex]:=dictblock.wordentry[dictindex];
  /*copy over dict x*/
  if dictblock.wordentry[dictindex] <> dictblock.endword then
    dictindex:=dictindex + 1
  else
    writeln;
  writeln('endword',
    dictblock.endword: stringsmax);
  write('numoldblock', numoldblock);
  writeln('of ', numberofblocks);

  numoldblock:=numoldblock + 1;
  if numoldblock <= numberofblocks then
    readblock(dictfile, numoldblock, dictblock)
  else
    clear(dictblock.startword);
  if dictblock.startword=emptyword then
begin
  finished:=true;
  tempblock.startword:=tempblock.wordentry[i];
  tempblock.endword:=
    tempblock.wordentry[tempindex];
  putblock(dictfile, tempblock, numnewblock);
  blocktable[numnewblock].startword:=
    tempblock.startword;
  blocktable[numnewblock].endword:=
    tempblock.endword
end
else dictindex:=1; /*new block*/
end
end
else
begin
  writeln('newword ', newword: stringsmax);
  writeln('before ', dictblock.wordentry[dictindex]: stringsmax);
  tempblock.wordentry[tempindex]:=newword;
  i:=i + 1;
  /*insert new word*/
  if i <= size then
    newword:=addarray[i]
  else clear(newword) /* just recopy rest of olddict */
else tempindex:=tempindex + 1;
if (tempindex > pagesize) then
begin
  tempindex:=1;
end
end
tempblock.startword:=tempblock.wordentry[1];
tempblock.endword:=tempblock.wordentry[pagenum];
putblock(dictfile, tempblock, numnewblock);
blocktable[numnewblock].startword:=
tempblock.startword;
blocktable[numnewblock].endword:=
tempblock.endword;
numnewblock:=numnewblock + 1;
if numnewblock > numberofblocks then
begin
writeLn('dictionary truncated');
finished:=true
end;
end;
writeblocktable;
close(dictfile)
end;

i:=0;
reset(addfile, addname, ch, 'add');

while not eof(addfile) and (i < pagesize) do
begin
i:=i + 1;
readfile(addfile, buffer);

if (buffer.ch[one] >= 'A') and (buffer.ch[one] <= 'Z')
then
begin
charact:=buffer.ch[one];  /* so 'ord' doesn't bomb */
temp:=ord(charact) + capoffset;
buffer.ch[one]:=chr[temp]
end;
addarray[i]:=buffer.ch;
end;
sortwords(addarray, i);
initadd(dictname, addarray[one], firstinsert);
insertwords(addarray, i, firstinsert, dictfile)
end;
procedure writeblocktable; external;

procedure putblock(dictfile : blockfile; dictblock : block; k : blockrange); external;

procedure delwords(delname, dictname : string);

var
dictfile: blockfile;
delarray : page;
delfile : text;
buffer : string;
firstcut, i : integer;

procedure clear(var s: chararray); external;

procedure writeblocktable;
external;

procedure readfile(fil : text; var s : string); /* for file input */

begin
  clear(s.ch);
s.len:=0;
with s do
  while (not eoln(fil)) and (len < stringmax) do
    begin
      len:=len + 1;
      read(fil,ch[len])
    end;
  readln(fil)
end;

procedure readblock(dictfile: blockfile; index: blockrange;
  var chosenblock : block); external;

procedure sortwords(var addarray : page; size : integer);
external;

procedure cutwords(delarray: page; delsize : integer;
  dictfile : blockfile);
```plaintext
var
dictindex, delindex, blockindex : integer;
dictblock : block;

begin
  delindex := 0;
  blockindex := 1;
  dictindex := 1;

  while (delindex < delsize) do
    begin
      delindex := delindex + 1;
      while (blocktable[blockindex].endword < delarray[delindex])
        do
          blockindex := blockindex + 1;
      readblock(dictfile, blockindex, dictblock);

      while (dictblock.wordentry[dictindex] <> delarray[delindex])
        and (dictblock.wordentry[dictindex] <> dictblock.endword)
        do
          dictindex := dictindex + 1;
          /* can't delete if not in dict */
          /* if word in dictionary do next section */

      if dictblock.wordentry[dictindex] = delarray[delindex]
        then
          begin
            if dictblock.wordentry[dictindex] = dictblock.endword then
              if dictindex <> one then
                begin
                  clear(dictblock.wordentry[dictindex]);
                  dictblock.endword := dictblock.wordentry[dictindex - 1];
                end
            else
              begin
                writeln(dictblock.wordentry[one], ' is the only word in ');
                writeln('this block. It cannot be deleted at this time.');
              end
          end
          else
            begin
              while (dictblock.wordentry[dictindex] + 1 <=
                dictblock.endword) do
                begin
                  dictblock.wordentry[dictindex] :=
                    dictblock.wordentry[dictindex + 1];
                  dictindex := dictindex + 1;
                end;
              clear(dictblock.wordentry[dictindex])
            end;
        end;
  endblocktable[blockindex].startword := dictblock.startword;
  blocktable[blockindex].endword := dictblock.endword;
  putblock(dictfile, dictblock, blockindex);
end; /*if dictblock*/
end /*while*/
end; /*cutwords*/
```
begin
  i:=0;
  reset(delfile, delname, ch, 'del');

  while not eof(delfile) and (i < pagesize) do
    begin
      i:=i + 1;
      readline(delfile, buffer);
      delarray[i] := buffer.ch;
    end;
  sortwords(delarray, i);

  reset(dictfile, dictname, ch, 'dict/seek');
  cutwords(delarray, i, dictfile);  /*cut it out of block*/
  writeblocktable
end;
procedure clear(var s : chararray); external;

procedure readstring(var s:string); /* works only for tty input */ external;

procedure initblocktable(tablefname: string); external;

procedure delwords(delname, dictname : string);
  external;

procedure addwords(addname, dictname : string);
  external;

begin
  clear(emptyword);

  writeln('enter tablefile name');
  readstring(tablefname);
  writeln('enter dictionary name in form file,ext/seek');
  readstrings(dictname);
  initblocktable(tablefname);
  writeln('do you want to add words to the dictionary? (y/n)');
  readln(inchar);
  if inchar='y' then
    begin
      writeln('enter addfile name');
      readstring(addfname);
      addwords(addfname, dictname)
    end;
  writeln('do you want to delete words from the dictionary? (y/n)');
  if inchar='y' then
    begin
      writeln('enter delname');
      readstrings(delname);
      delwords(delname, dictname)
    end;
end.
APPENDIX D

CONSTITUENT PROGRAMS OF GENDCT.SAV
GENDCT
Major Procedures

- CLEAR
- READSTRING
- READFILE
- GENDICTFILE
- WRITEBLOCKTABLE

DLIST
Major Procedures

- CLEAR
- READSTRING
- INITBLOCKTABLE
- LISTDCT
  - READBLOCK
Procedure writeBlocktable;

    type
cfile=file of chararray;

    var
tablefile : cfile;
index : integer;
tablename : strings;

Procedure clear(var s : chararray);
    var i: integer;
begin
    for i:=1 to stringsmax do s[i]:=blank
end; /*clear*/

Procedure readstring(var s:string); /*for tty input*/

begin
    clear(s.ch);
s.len:=0;
with s do
    while (not eoln) and (len < stringsmax) do
    begin
        len:=len + 1;
        read(ch[len])
    end;
    readln
end; /*readstring*/

begin
/*writes pairs of entries in blocktable as
chararrays, not as entryInBlocktable, which
includes pointer*/

writeln('enter tablefile name');
readstring(tablename);
rewrite(tablefile, tablename.ch, 'tab');
for index:=1 to numberofblocks do
begin
    tablefile^:=blocktable[index].startword;
    put(tablefile);
    tablefile^:=blocktable[index].endword;
    put(tablefile)
end;
clear(blocktable[index].startword); /*blank marks end*/
    tablefile^:=blocktable[index].startword;
    put(tablefile);
    tablefile^:=blocktable[index].startword;
    put(tablefile);
close(tablefile)
end;
procedure clear(var s: chararray);
  var i: integer;
  begin
    for i:=1 to stringmax do s[i]:='
  end;

procedure readFile(infil: text; var s:string); /*for file input*/
begin
clear(s.ch);
s.len:=0;
with s do
  while not eoln(infil) and (len < stringmax) do
    begin
      len:= len + 1;
      read(infil, ch[len])
      end;
readln(infil)
end;

procedure readstrings(var s:string); /*for tty input*/
begin
clear(s.ch);
s.len:=0;
with s do
  while (not eoln) and (len < stringmax) do
    begin
      len:= len + 1;
      read(ch[len])
      end;
readln
end;

*****************************************************************************

procedure sendictfile;

  type
dfile=file of block;

  var
dictfile : dfile;
infname, dictname, precedingword, presentword : string;
dictblock : block;
numphysblock, index, i : integer;
infil : text;

begin
writeln('generating dictionary, enter dictionary name');
readstring(dictname);

numphysblock:=(numofblocks * (((pagesize + 2) * stringmax) div 512));
*so RT1 allocates sufficient space for dict*

```c
writeln('rewriting...');
rewrite(dictfile, dictname.ch, 'dct', numphyblock);
writeln('rewrite done');
writeln('enter input file name');
readstring(infname);
writeln('resetting...');
reset(infil, infname.ch);
writeln('reset done');
readfile(infil, presentword);
dictblock, wordentry[one] = presentword, ch;
presentword := presentword; /* have to keep track of order, as we don't sort */
index := 0;
i := 1;

with dictblock do

while (index < numberofblocks) and (not eof(infil)) do begin
    index := index + 1;
    repeat
        begin
            readFile(infil, presentword);
            if presentword.ch > precedingword.ch then begin
                i := i + 1;
                wordentry[i] := presentword.ch;
                precedingword := presentword
            end
        else
            writeln('word rejected, out of order', presentword.ch; stringmax)
    end
    until ((i = pagesize) or eof(infil))
startword := wordentry[one];
endword := precedingword.ch; /* always true */
dictfile := dictblock; /* is this OK with 'with' */
put(dictfile);
blocktable[index].startword := dictblock, startword;
blocktable[index].endword := dictblock, endword;
i := 0
end; /* while */ /* cycle for one block */

while index < numberofblocks do begin
    index := index + 1;
clear(dictblock.startword);
clear(dictblock.endword); /* unfilled blocks blank */
    for i := 1 to pagesize do
        clear(dictblock, wordentry[i]);

dictfile := dictblock;
put(dictfile);
blocktable[index].startword := dictblock, startword;
blocktable[index].endword := dictblock, endword
end;
close(dictfile)
end
```
Procedure writeblocktable;

external;

begin
  sendictfile;
  writeblocktable
end.
APPENDIX E

CONSTITUENT PROGRAM OF DLIST.SAV
procedure clear(var s: chararray); external;
procedure readstrings(var s: string); external;
procedure initblocktable(tablename: string); external;
procedure readblock(dictfile: blockfile; index: blockrange; var chosenblock: block);
   external;
procedure listdct(dictname, outfname: string);

var
dictfile: blockfile;
outfile: text;
dictblock: block;
index, r: integer;
buf1, buf2: chararray;

begin
reset(dictfile, dictname.ch, 'dct/seek');
reset(outfile, outfname.ch, '1st');
index! = 0;
repeat
   begin
      index! = index + 1;
      readblock(dictfile, index, dictblock);
      buf1 := blocktable[index].startword;
      buf2 := dictblock.startword;
      writeln(outfile);
      writeln(outfile, 'blocktable.startword ',
         buf1 :stringmax, 'dictblock.startword ',
         buf2 :stringmax);
      buf1 := blocktable[index].endword;
      buf2 := dictblock.endword;
      writeln(outfile, 'blocktable.endword ',
         buf1 :stringmax, 'dictblock.endword ',
         buf2 :stringmax);
      writeln(outfile);
      j! = 0;
      repeat
         begin
            j! = j + 1;
            if (j mod 4) = 0 then
               writeln(outfile, dictblock.wordentry[j])
            else
               write(outfile, dictblock.wordentry[j])
         end
      until dictblock.wordentry[j] = dictblock.endword
      until blocktable[index].startword = emptyword
   end;
begin
clear(emptyword);
writeln('enter listings file name');
readstring(outfilename);
writeln('enter tablefile name');
readstring(tablefname);
writeln('enter dictionary name in form file.ext/seek');
readstring(dictname);
initblocktable(tablefname);
listdct(dictname, outfilename)
end.