5-30-1986

SPLCHK: A VAX/VMS spelling checker and corrector utilizing a trie-structured dictionary

Roman George Lewyckyj

Rochester Institute of Technology
School of Computer Science and Technology

SPLCHK
A VAX/VMS Spelling Checker and Corrector Utilizing a Trie-structured Dictionary

by
Roman George Lewyckyj

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Peter G. Anderson
Dr. Peter G. Anderson

John A. Biles
Prof. John A. Biles

Rayno Niemi
Dr. Rayno Niemi

May 30, 1986

930120

Permission to Reproduce

Title of Thesis: SPLCHK, A VAX/VMS Spelling Checker and Corrector Utilizing a Trie-structured Dictionary

I, Roman Lewyckyj, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 1/18/88

Roman Lewyckyj

1 PRELIMINARY INFORMATION

1.1 Abstract

This paper presents a "user-friendly" spelling checker and corrector written in C, which utilizes a "trie" structure to store its main dictionary of words, thereby eliminating redundancy of letters. The program allows the user to either check the spelling of words or files, or to correct "misspelled" words in files. The program is designed to operate primarily in a minicomputer environment, such as the VAX/VMS environment.

1.2 Key Words And Phrases

Spelling programs, spelling errors, spelling correction, typographical errors, spelling dictionary, dictionary lookup, searching, data compression, trie, information storage and retrieval

1.3 Computing Review Subject Codes

Primary classification code:

H.3.1 Content Analysis and Indexing

Secondary classification codes:

I.7.1 Text Editing

H.3.3 Information Search and Retrieval

H.4.1 Office Automation

1.4 Table Of Contents

TABLE OF CONTENTS

2 INTRODUCTION AND BACKGROUND

2.1 Problem Statement

The area explored in this thesis is that of computer-based detection and correction of spelling errors in various types of text files (e.g. documents, reports, memos, letters, possibly programs), specifically in a multi-user, multi-tasking minicomputer environment such as VAX/VMS. Several programs designed for just this purpose already exist, and are in use on various computer systems.

However, there are several general problems that exist in this field, and many of the current spelling checkers display at least one of these problems. Among these problems are:

(A) Large, cumbersome dictionary files, which lead to slow access to the data contained in these files, which in turn ends up slowing down the overall performance of the spelling checker.

(B) Programs which produce only a display of incorrectly spelled words, and do not allow for interactive correction of these misspellings.

(C) Algorithms for generating correct spelling guesses which produce unacceptable results (i.e. do not produce the intended word in their list of possible replacements).

(D) Output/functionality which is not user-friendly, such as a spelling checker which produces a list of incorrectly spelled words without any context for those words, such as line numbers, surrounding text, etc; or a spelling checker which does not have a consistent user interface across its various functions.

(E) Hashing functions (where used) which do not give acceptable results, such as cases in which correctly spelled words are flagged as misspellings, or in which misspelled words are accepted as correctly spelled.

2.2 Previous Work

Some of the earliest work in this field dates back to the late fifties and early sixties. From beginnings in the areas of OCR (Optical Character Recognition), correction of data entry errors, correction of data base information errors, keyword recognition, etc., we have come to the point where we have programs that not only check for spelling errors, but also (in many cases, and with some user interaction) correct these errors once they are found.

Along the way much work was done in the related areas of string matching and correction algorithms, and efficient space-use algorithms, leading to the current sophisticated programs for the correction of misspelled input.

In general most spelling checkers/correctors today fall into one of three main categories: those that use some type of dictionary file to look up the word in question, those that use one or more hashing functions to decide whether or not the word in question is correctly spelled, and those that look at digram (two-letter pairs) and/or trigram (three-letter triples) frequencies to decide whether a word is likely to be correctly spelled.

An example of this first category is the spelling checker written and described by Peterson [16,17], upon which many of the SPLCHK concepts are based. Peterson utilizes a three-tiered search strategy by first checking an input word against a small static table of common words, then against a fairly small, document- or user-specific list of words, and finally against a large dictionary file. If a word is not located during any of these three searches, then an algorithm based on common misspellings (see Section 2.3) is applied to the word, and each of the possible candidates is in turn verified, with those verified by one of the three searches presented to the user as possible correct spellings of the misspelled word.

Another example of the first category of spelling checkers is the program written and described by Robinson and Singer [18]. This program breaks down the source text into a tree containing at each node a distinct word, along with initial occurrence line number and total occurrence count. This tree is then traversed and compared with two alphabetically organized dictionary files,

the first a system-wide dictionary, the second a user-specific dictionary. Words not located in the dictionary are then displayed to the user, along with line and count information. Among the user options at this point are: mark the word as incorrect, correct the spelling of the word, accept the word as correctly spelled and add it to the second dictionary, and accept the word as correctly spelled without adding it to the dictionary.

This category of spelling checkers normally sacrifices some efficiency in the area of speed and size in return for a definitive conclusion as to the correctness of spelling.

An example of the second category of spelling checkers, those utilizing hashing functions, is the spelling checking portion of the "Z" screen editor used at Yale University [14]. This spelling checker is integrated into the screen editor, and is invoked by a single keystroke. It is based on passing input text through one or more hashing functions, and deciding the validity of the spelling based on the output of the hashing. Spelling checkers in this category are efficient in terms of speed and size, at the cost of being somewhat probabilistic (misspelled words can be accepted as correctly spelled).

An illustration of this method is presented by Nix [14] in the example of checking the spelling of a word against a dictionary of 1000 words. This dictionary is represented as a 20,000 element bit table, and is accessed through ten independent hashing functions. Initially, the bit table is all zeros. Each

of the 1000 words turns on ten bits as it is added. Spelling is checked by hashing the input word through the ten functions. If any of the bits is zero, the word is not in the dictionary. If all of the bits are ones, the word is assumed to be in the dictionary. Although misspelled words can rarely be accepted as correct, the author states that the probability of this occurring is less than 0.1 percent.

An example of the third category of spelling checkers, those looking at digram and/or trigram frequencies, is the TYPO program on the UNIX system [16]. The TYPO program computes the digram and trigram frequencies of input text and generates a list of the distinct tokens in the text. This list of tokens is then used to compute an "index of peculiarity," which is a measure of how well a particular token fits into the context of the rest of the input text. TYPO outputs the list of tokens sorted by this index of peculiarity. Misspelled words are usually uncovered because this list is relatively short, and because words with the highest index of peculiarity (the most likely misspellings) are at the top of the list [16].

This method is also efficient in terms of speed and size, but this time at the expense of not being able to actually classify a word as correctly spelled or misspelled, but rather only as fitting or not fitting the context of the rest of the text, and then leaving the user to decide if the word is indeed misspelled or not.

2.3 Theoretical And Conceptual Development

Many of today's sophisticated spelling checkers are not perfect, and leave room for improvement in one or more of the areas mentioned previously.

The main goals of the SPLCHK spelling error detector and corrector described in this thesis are the achievement of an efficient combination of size, speed, and user-friendliness.

The trie-structured dictionary is the basis for the elimination of many of the redundancies involved in other dictionary storage methods, as well as for the ability to access and search a dictionary in a reasonably efficient manner. The concept of user-friendliness is the basis for the SPLCHK program presenting a consistent and straightforward user interface, and being helpful to its user in the correction of misspellings, by making it as easy as possible to correct spelling errors with a minimal amount of user interaction.

Based on the research into previous work in the field of spelling error detection and correction (see Section 2.2), several major concepts and theories were incorporated into the SPLCHK program. Among these are the following:

(A) A trie-structured dictionary. Based on the concept of elimination of redundancy in information retrieval, a trie [21] is a specialized form of a tree. By storing the letters of a word in the nodes along a path in the trie,

the retrieval of that path will produce the spelling of that particular word. Once letters have been placed in the trie, they can be used in any number of paths of partially similar words; branches in the trie occur where the letters of the words differ. In the example in Figure 1 at the end of this section, the words "be", "bee", "beef", "beet", and "beets" would require eighteen characters if stored as separate words in a dictionary. However, using a trie structure, they can be stored in eleven characters, since the first two letters, "b" and "e", are common in all five words, and the second "e" and the "t" are common in more than one of the words. Therefore, the eleven characters that would need to be stored are "b", "e", and a word terminator for "be", a second "e" and a word terminator for "bee", an "f" and a word terminator for "beef", a "t" and a word terminator for "beet", and an "s" and a word terminator for "beets". The number of characters saved is even greater as the number of words stored is greater, and the redundancy in letters among words increases.

(B) The use of two (or three) "dictionary" structures. As demonstrated in Peterson's spelling checker [16,17], this concept involves the use of more than one type of structure to contain various portions of the "dictionary."

The first structure against which words would be checked is an unchanging list of perhaps one to three hundred of the most commonly used words in the English language. Since most of these commonly used words are usually two to four letters long, this list will not be very large in terms of space per word. Also, since the words in this list have a very high comparative frequency of use, many dictionary searches will be eliminated at this point.

The second search would involve tables of unigrams, digrams, and trigrams, against which all the one, two, and three letter combinations in a word, respectively, would be checked. Although successfully locating all of these combinations in the three tables would not insure that a word was correctly spelled, if any of the combinations was not found in its respective table, the word would be known to be incorrectly spelled.

The third step in this search sequence would finally be a search of the master trie-structured dictionary, but only in cases where neither of the previous two searches turned up a resolution as to the correctness of spelling.

(C) A user-friendly checking and correction environment. The degree of user friendliness displayed by a spelling checker program will directly affect the amount of use people get from the program, and also the amount of use

the program gets from users. For example, a system which simply gives a list of misspelled words with no other information will be used only if no other spelling checker is available.

Various ideas on user-friendliness in a spelling error detection and correction program exist; they involve the display of misspelled words "in context" (i.e. with line numbers, surrounding text, etc.) to give the user some information about the particular word; the use of program call parameters to allow the user to access particular features of the spelling checker as he or she wishes; and a listing of possible substitutes for misspelled words, along with the ability to interactively correct misspellings in files.

The SPLCHK program utilizes all of these features and, with the different options and features available, can be used by almost any type of computer user, from novice to expert.

(D) Common misspellings of words - based on Damerau's findings regarding different types of word misspellings [4], the kinds of errors shown in the following list account for about eighty percent of all common spelling mistakes:

(1) a transposition of two letters

(2) one extra letter

(3) a missing letter

(4) one wrong letter

As mentioned by Peterson [17], a typical correction algorithm involves generating a list of all possible words that could produce the incorrect word using the following rules:

(1) For case (1) above, all pairs of letters in the word in turn are transposed.

(2) For case (2) above, a letter is removed in turn from each position in the word.

(3) For case (3) above, each letter of the alphabet is in turn inserted before and/or after each letter in the word.

(4) For case (4) above, each letter of the alphabet is in turn substituted for each letter in the word.

These potential corrections are then checked against the dictionary and, if found there, are shown to the user, as either a most obvious choice if there is only one candidate, or as a list of possible choices if there is more than one guess. This is the algorithm which was

implemented for the file correction portion of the SPLCHK spelling checker.

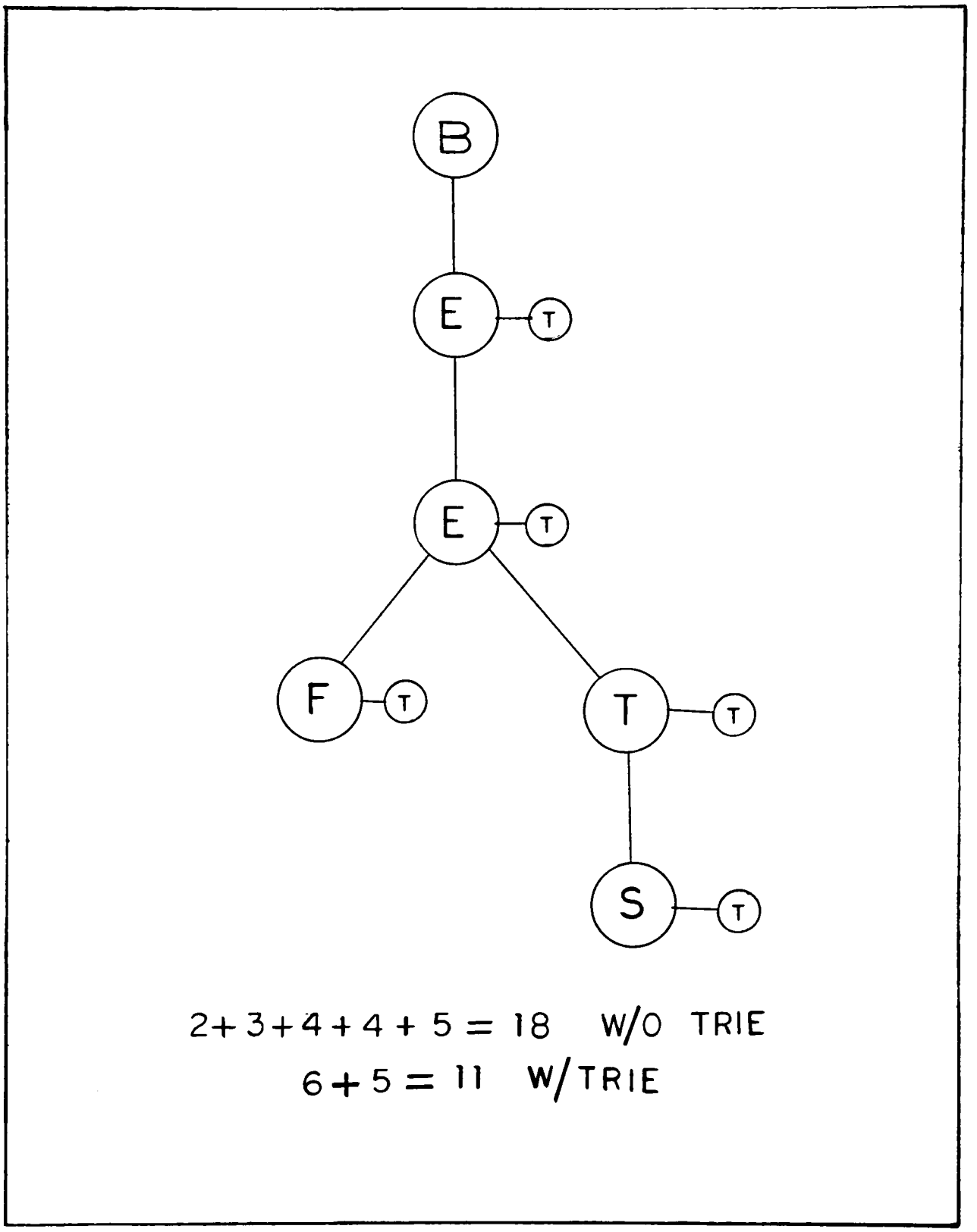


Figure 1

3 FUNCTIONAL SPECIFICATION

- Functions to be Performed by the SPLCHK Program

(A) The SPLCHK program will store the letters of (correctly-spelled) English language words in a trie-structure dictionary file.

(B) In dictionary-create mode, the SPLCHK program will allow an "authorized" user (again, based on access to the dictionary file and its directory) to create a new dictionary file, and to add words to it.

(C) In dictionary-update mode, the SPLCHK program will allow an "authorized" user (based on access to the dictionary file) to "delete" incorrect words from the dictionary file, or to "modify" misspelled words in the dictionary.

(D) In list contents mode, the SPLCHK program will allow a user to list out the words that are currently stored in the program's dictionary.

(E) In word-check mode, the SPLCHK program will compare words entered by the user from the keyboard against the words in the dictionary, and indicate to the user whether or not the word in question is correctly spelled and, if the spelling is incorrect, will display a list of possible correct spellings.

(F) In file-check mode, the SPLCHK program will compare words from a user's input file against those in its dictionary, indicating to the user any misspelled or unknown words located in the file.

(G) In file-correction mode, the SPLCHK program will compare words from a user's input file against those in its dictionary. The program will copy any correct input words as they are to the output file, while displaying misspelled or incorrect words "in context"; i.e., with some lines of surrounding text and lines numbers, along with a list of possible correct spellings for the word, and allowing the user to accept one of the SPLCHK-generated guesses, to enter a new spelling for the word, or to accept the word as is. The user-selected word will then be copied to the output file. The output file will be sequentially structured.

(H) In help mode, the SPLCHK program will display on-line help information, briefly describing how to use of the program.

- Limitations and Restrictions of the SPLCHK program

(A) The SPLCHK program will not signal errors in cases in which a correctly spelled word is used in a grammatically or syntactically incorrect manner, or is used in the wrong context.

(B) The SPLCHK program will not necessarily generate the correct choice for replacing a misspelled word, depending on the degree of incorrectness of the original spelling.

(C) The SPLCHK program will not currently accept a user-specific or secondary dictionary file.

(D) The file-correction mode of the SPLCHK program does not have a "log file" option, as was described in the proposal. Because of the interactive nature of file correction, the need for or usefulness of a log file did not seem to be justified.

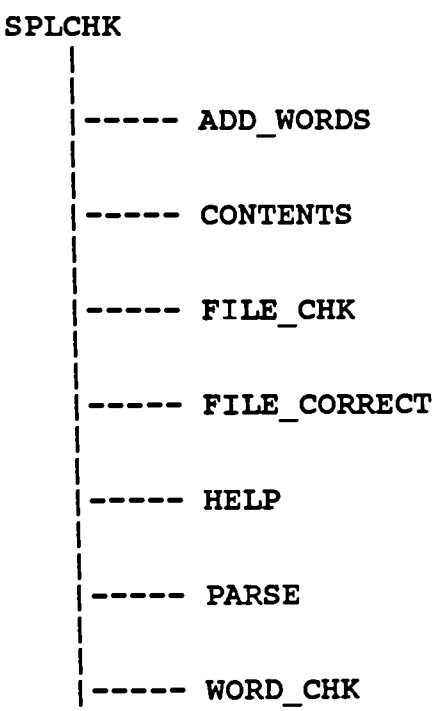
(E) The SPLCHK program does not currently check for "authorized" users in dictionary creation or update modes, or in dictionary listing mode. However, to avoid users creating or modifying dictionaries, the SPLCHK dictionary and informational file should be placed in a directory which allows only read access to non-privileged system users.

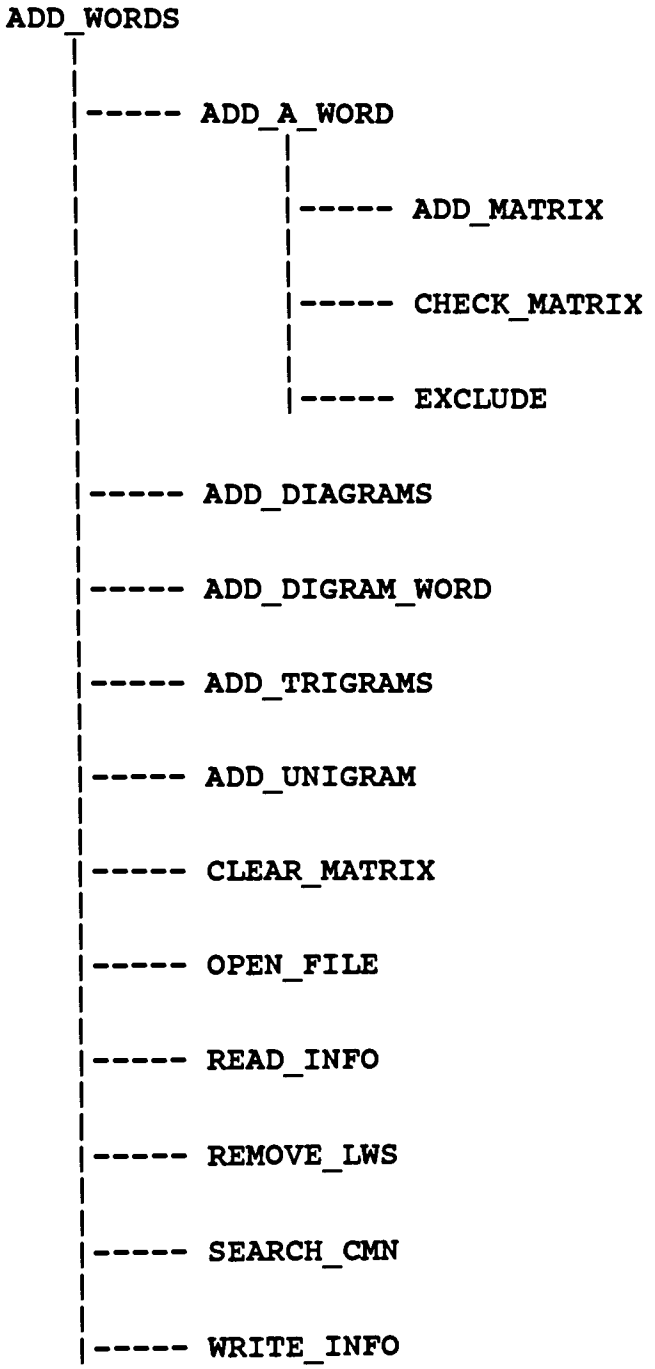
(F) The SPLCHK program does not allow "real" word deletion or modification; this process is currently done by generating a listing of the dictionary, deleting or modifying the words in that listing, and then regenerating the dictionary from that updated listing.

(G) Interruption of the dictionary update process will most likely corrupt the dictionary and informational files. Therefore, prior to adding words to the SPLCHK dictionary, backup copies of the dictionary and informational files should be created.

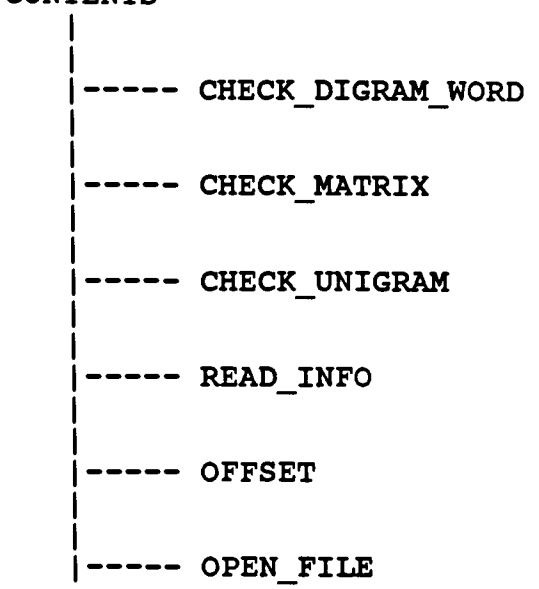
4 ARCHITECTURAL DESIGN

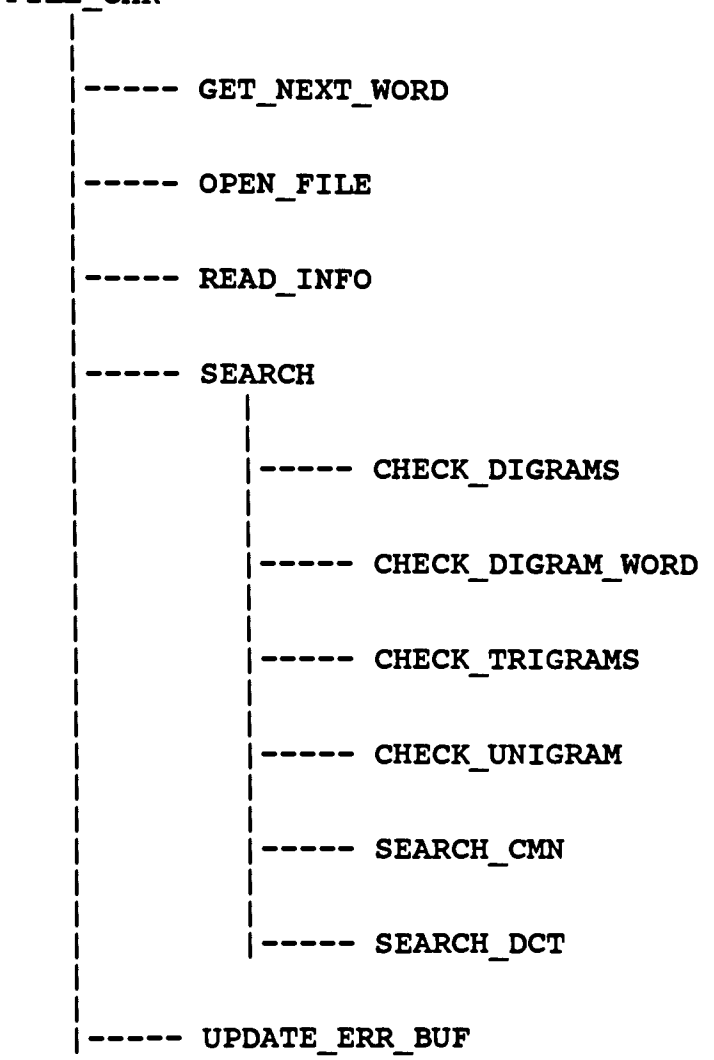
4.1 Structure Charts

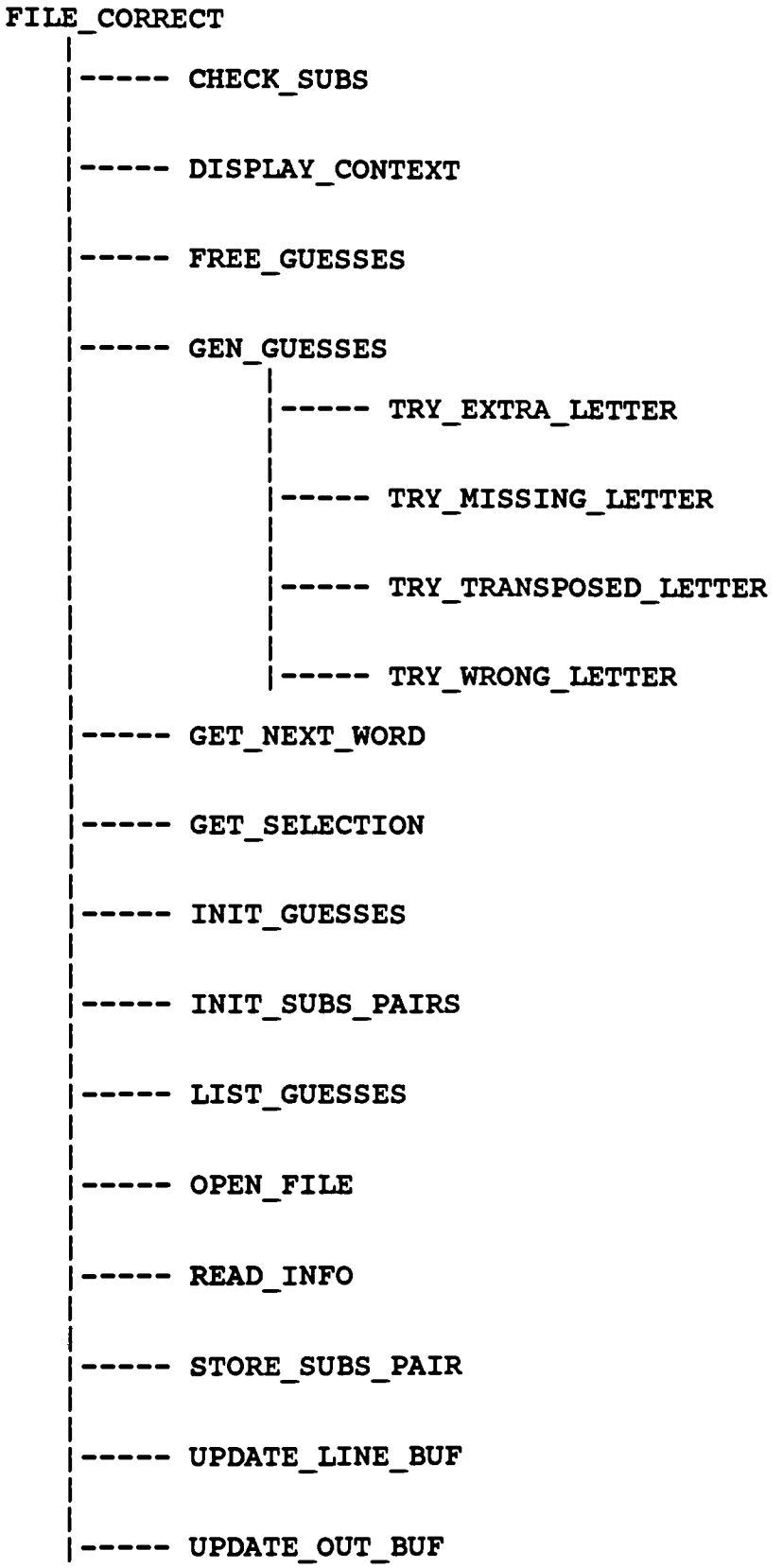


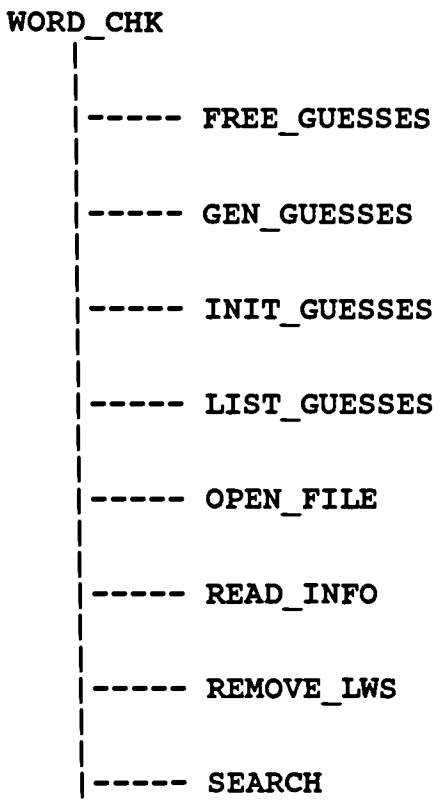

ADD_WORDS
ADD_A_WORD
ADD_MATRIX
CHECK_MATRIX
EXCLUDE
ADD_DIAGRAMS
ADD_DIGRAM_WORD
ADD_TRIGRAMS
ADD_UNIGRAM
CLEAR_MATRIX
OPEN_FILE
READ_INFO
REMOVE_LWS
SEARCH_CMN
WRITE_INFO

CONTENTS


CHECK_DIGRAM_WORD
CHECK_MATRIX
CHECK_UNIGRAM
READ_INFO
OFFSET
OPEN_FILE


FILE_CHK
GET_NEXT_WORD
OPEN_FILE
READ_INFO
SEARCH
CHECK_DIGRAMS
CHECK_DIGRAM_WORD
CHECK_TRIGRAMS
CHECK_UNIGRAM
SEARCH_CMN
SEARCH_DCT
UPDATE_ERR_BUF


FILE_CORRECT
CHECK_SUBS
DISPLAY_CONTEXT
FREE_GUESSES
GEN_GUESSES
TRY_EXTRA_LETTER
TRY_MISSING_LETTER
TRY_TRANSPOSED_LETTER
TRY_WRONG_LETTER
GET_NEXT_WORD
GET_SELECTION
INIT_GUESSES
INIT_SUBS_PAIRS
LIST_GUESSES
OPEN_FILE
READ_INFO
STORE_SUBS_PAIR
UPDATE_LINE_BUF
UPDATE_OUT_BUF


WORD_CHK
FREE_GUESSES
GEN_GUESSES
INIT_GUESSES
LIST_GUESSES
OPEN_FILE
READ_INFO
REMOVE_LWS
SEARCH

4.2 Pseudocode

```
BEGIN_PROGRAM
Get COMMAND_LINE from program call
{ Parse command line for QUALIFIERS and FILENAMES }
WHILE (not end of COMMAND_LINE)
{ QUALIFIERS }
  IF (current char is a "/") THEN
    Increment current char
    Index into QUALIFIERS based on current char
    IF (char found in QUALIFIERS) THEN
      Increment current char
    END_IF
    WHILE (current char is not COMMAND_DELIM)
      Increment current char
      Compare current char with next char in QUALIFIERS row
    END_WHILE
    IF (all chars compare with QUALIFIERS row) THEN
      Set appropriate flag in MODES_SET
    ELSE
      Display appropriate error message
      END_PROGRAM
    END_IF
{ FILENAMES }
  ELSE_IF (current char is not COMMAND_DELIM) THEN
    WHILE (current char is not COMMAND_DELIM)
```

```
      Store current char into record in FILE_LIST
      Increment current char
    END_WHILE
{ DELIMITERS }
  ELSE { current char is COMMAND_DELIM }
    Increment current char
  END_IF
END_WHILE


{ CHECK INPUT FILES }
IF (MODES_SET includes FILE_CHECK) THEN
  IF (no filenames in FILE_LIST) THEN
    Prompt user for file(s) to be processed
    IF (filename(s) inputted by user) THEN
      Store filename(s) into record(s) in FILE_LIST
    ELSE { no filename(s) inputted by user }
      END_PROGRAM
    END_IF
  END_IF
  WHILE (more files to process)
    Open current input file for reading only
    IF (input file open successful) THEN
      Open new sequential output file for writing
      Read first record and attach DISPLAY_LINES pointer 4
      Read second record and attach DISPLAY_LINES pointer 5
      WHILE (not end of current input file plus final two records)
```

```
Read next record from input file into INPUT_BUFFER
Attach DISPLAY_LINES pointer 1 to pointer 2's record
Attach DISPLAY_LINES pointer 2 to pointer 3's record
Attach DISPLAY_LINES pointer 3 to pointer 4's record
Attach DISPLAY_LINES pointer 4 to pointer 5's record
Attach DISPLAY_LINES pointer 5 to record read
WHILE (not end of CURRENT_RECORD)
  Update CURRENT_WORD_BEGIN pointer
  Check current char against WORD_DELIM
  IF (current char is not delimiter) THEN
    WHILE (delimiter not encountered)
      Increment current char
    END_WHILE
    Update CURRENT_WORD_END pointer
  ELSE { current char is delimiter }
    WHILE (current char is delimiter)
      Copy char to OUTPUT_BUFFER
      Increment current char
    END_WHILE
  END_IF
  Copy word to be checked into CURRENT_WORD
  Convert CURRENT_WORD to uppercase
  Search SUBS_PAIRS for CURRENT_WORD
  IF (CURRENT_WORD not found in SUBS_PAIRS) THEN
    Check CURRENT_WORD against COMMON_WORDS
    IF (CURRENT_WORD not found in COMMON_WORDS) THEN
      Use DIGRAM_MATRIX to index into DICTIONARY
```

```
            Search DICTIONARY for CURRENT_WORD
            IF (CURRENT_WORD not found in DICTIONARY) THEN
              Display incorrectly spelled CURRENT_WORD in context
              Ask user if he wants correct-spelling guesses
              IF (answer is yes) THEN
{ TRY TRANSPOSED LETTERS }
                WHILE (more letters in CURRENT_WORD)
                  Exchange current letter with next letter
                  Check new word against LEGAL_DIGRAMS
                  IF (all digrams in new word are legal) THEN
                    Check new word against COMMON_WORDS
                    IF (new word found in COMMON_WORDS) THEN
                      Store new word in GUESS_LIST
                    ELSE { new word not in COMMON_WORDS }
                      Use DIGRAM_MATRIX to index into DICTIONARY
                      Search DICTIONARY for CURRENT_WORD
                      IF (new word found in DICTIONARY) THEN
                        Store new word in GUESS_LIST
                      END_IF
                    END_IF
                  END_IF
                  Increment current letter
                END_WHILE
{ TRY EXTRA LETTER }
                WHILE (more letters in CURRENT_WORD)
                  Remove current letter
                  Check new word against LEGAL_DIGRAMS
```

```
                    IF (all digrams in new word are legal) THEN
                      Check new word against COMMON_WORDS
                      IF (new word found in COMMON_WORDS) THEN
                        Store new word in GUESS_LIST
                      ELSE { new word not in COMMON_WORDS }
                        Use DIGRAM_MATRIX to index into DICTIONARY
                        Search DICTIONARY for CURRENT_WORD
                        IF (new word found in DICTIONARY) THEN
                          Store new word in GUESS_LIST
                        END_IF
                      END_IF
                    END_IF
                    Increment current letter
                  END_WHILE
{ TRY WRONG LETTER }
                  WHILE (more letters in CURRENT_WORD)
                    WHILE ( more letters of the alphabet)
                      Replace current word letter with current -
                       alphabet letter
                      Check new word against LEGAL_DIGRAMS
                      IF (all digrams in new word are legal) THEN
                        Check new word against COMMON_WORDS
                        IF (new word found in COMMON_WORDS) THEN
                          Store new word in GUESS_LIST
                        ELSE { new word not in COMMON_WORDS }
                          Use DIGRAM_MATRIX to index into DICTIONARY
                          Search DICTIONARY for CURRENT_WORD
```

```
                    IF (new word found in DICTIONARY) THEN
                      Store new word in GUESS_LIST
                    END_IF
                  END_IF
                END_IF
                Increment current alphabet letter
              END_WHILE
              Increment current word letter
            END_WHILE
{ TRY MISSING LETTER }
            WHILE (more letters in CURRENT_WORD)
              WHILE (more letters of the alphabet)
                Insert current alphabet letter between -
                 current word letter and next word letter
                Check new word against LEGAL_DIGRAMS
                IF (all digrams in new word are legal) THEN
                  Check new word against COMMON_WORDS
                  IF (new word found in COMMON_WORDS) THEN
                    Store new word in GUESS_LIST
                  ELSE { new word not in COMMON_WORDS }
                    Use DIGRAM_MATRIX to index into DICTIONARY
                    Search DICTIONARY for CURRENT_WORD
                    IF (new word found in DICTIONARY) THEN
                      Store new word in GUESS_LIST
                    END_IF
                  END_IF
                END_IF
```

```
                    Increment current alphabet letter
                  END_WHILE
                  Increment current word letter
                END_WHILE
                Display contents of GUESS_LIST to user
              END_IF
              Prompt user for a selection from GUESS_LIST or a -
               substitute for CURRENT_WORD
              Copy user selection or input to OUTPUT_BUFFER
              Add CURRENT_WORD and substitute to SUBS_PAIRS
            ELSE { CURRENT_WORD found in DICTIONARY }
              Copy CURRENT_WORD to OUTPUT_BUFFER
            END_IF
          ELSE { CURRENT_WORD found in COMMON_WORDS }
            Copy CURRENT_WORD to OUTPUT_BUFFER
          END_IF
        ELSE { CURRENT_WORD found in SUBS_PAIRS }
          Copy substitute from SUBS_PAIRS into OUTPUT_BUFFER
        END_IF
      END_WHILE
      Write OUTPUT_BUFFER to output file
    END_WHILE
    Close input file
    Close output file
  ELSE { input file open unsuccessful }
    Display error message regarding unsuccessful file open
  END_IF
```

```
    Get next filename from FILE_LIST
  END_WHILE


{ INTERACTIVE INPUT FROM USER }
ELSE_IF (MODE is WORD_CHECK) THEN
  WHILE (not end of user input)
    Prompt user for input to be checked
    IF (not end of user input) THEN
      Accept input into CURRENT_RECORD
      WHILE (not end of CURRENT_RECORD)
        Update CURRENT_WORD_BEGIN pointer
        Check current char against WORD_DELIM
        IF (current char is not delimiter) THEN
          WHILE (delimiter not encountered)
            Increment current char
          END_WHILE
          Update CURRENT_WORD_END pointer
        ELSE { current char is delimiter }
          WHILE (current char is delimiter)
            Copy char to OUTPUT_BUFFER
            Increment current char
          END_WHILE
        END_IF
        Copy word to be checked into CURRENT_WORD
        Convert CURRENT_WORD to uppercase
        Check CURRENT_WORD against COMMON_WORDS
```

```
        IF (CURRENT_WORD not found in COMMON_WORDS) THEN
          Use DIGRAM_MATRIX to index into DICTIONARY
          Search DICTIONARY for CURRENT_WORD
          IF (CURRENT_WORD not found in DICTIONARY) THEN
{ TRY TRANSPOSED LETTERS }
            WHILE (more letters in CURRENT_WORD)
              Exchange current letter with next letter
              Check new word against LEGAL_DIGRAMS
              IF (all digrams in new word are legal) THEN
                Check new word against COMMON_WORDS
                IF (new word found in COMMON_WORDS) THEN
                  Store new word in GUESS_LIST
                ELSE { new word not in COMMON_WORDS }
                  Use DIGRAM_MATRIX to index into DICTIONARY
                  Search DICTIONARY for CURRENT_WORD
                  IF (new word found in DICTIONARY) THEN
                    Store new word in GUESS_LIST
                  END_IF
                END_IF
              END_IF
              Increment current letter
            END_WHILE
{ TRY EXTRA LETTER }
            WHILE (more letters in CURRENT_WORD)
              Remove current letter
              Check new word against LEGAL_DIGRAMS
              IF (all digrams in new word are legal) THEN
```

```
                Check new word against COMMON_WORDS
                IF (new word found in COMMON_WORDS) THEN
                  Store new word in GUESS_LIST
                ELSE { new word not in COMMON_WORDS }
                  Use DIGRAM_MATRIX to index into DICTIONARY
                  Search DICTIONARY for CURRENT_WORD
                  IF (new word found in DICTIONARY) THEN
                    Store new word in GUESS_LIST
                  END_IF
                END_IF
              END_IF
              Increment current letter
            END_WHILE
{ TRY WRONG LETTER }
            WHILE (more letters in CURRENT_WORD)
              WHILE ( more letters of the alphabet)
                Replace current word letter with current alphabet -
                 letter
                Check new word against LEGAL_DIGRAMS
                IF (all digrams in new word are legal) THEN
                  Check new word against COMMON_WORDS
                  IF (new word found in COMMON_WORDS) THEN
                    Store new word in GUESS_LIST
                  ELSE { new word not in COMMON_WORDS }
                    Use DIGRAM_MATRIX to index into DICTIONARY
                    Search DICTIONARY for CURRENT_WORD
                    IF (new word found in DICTIONARY) THEN
```

```
                    Store new word in GUESS_LIST
                  END_IF
                END_IF
              END_IF
              Increment current alphabet letter
            END_WHILE
            Increment current word letter
          END_WHILE
{ TRY MISSING LETTER }
          WHILE (more letters in CURRENT_WORD)
            WHILE (more letters of the alphabet)
              Insert current alphabet letter between current -
               word letter and next word letter
              Check new word against LEGAL_DIGRAMS
              IF (all digrams in new word are legal) THEN
                Check new word against COMMON_WORDS
                IF (new word found in COMMON_WORDS) THEN
                  Store new word in GUESS_LIST
                ELSE { new word not in COMMON_WORDS }
                  Use DIGRAM_MATRIX to index into DICTIONARY
                  Search DICTIONARY for CURRENT_WORD
                  IF (new word found in DICTIONARY) THEN
                    Store new word in GUESS_LIST
                  END_IF
                END_IF
              END_IF
              Increment current alphabet letter
```

```
              END_WHILE
              Increment current word letter
            END_WHILE
            Display incorrectly spelled word to user
            Display contents of GUESS_LIST to user
          END_IF
        END_IF
      END_WHILE
    END_IF
  END_WHILE


{ PRINT CONTENTS OF DICTIONARY FILE }
ELSE_IF (MODE is LIST_WORDS) THEN
  Open new sequential output file for writing
  IF (output file open successful) THEN
    WHILE (more rows in LEGAL_DIGRAMS)
      WHILE (more columns in LEGAL_DIGRAMS)
        Use same row and column from DIGRAM_MATRIX to index into -
         DICTIONARY
        WHILE (more CHILD_POINTERs) OR (more SIBLING_POINTERS)
          WHILE (more CHILD_POINTERs)
            Add char to OUTPUT_BUFFER
            IF (TERMINATOR) THEN
              Write OUTPUT_BUFFER to output file
            END_IF
            Follow CHILD_POINTER
```

```
          END_WHILE
          IF (SIBLING_POINTER exists) THEN
            Follow SIBLING_POINTER
          ELSE { no SIBLING_POINTER }
            WHILE (no SIBLING_POINTERs)
              Back up one level to the PARENT record using -
               LIST_POINTERS
              Erase last character from OUTPUT_BUFFER
            END_WHILE
          END_IF
        END_WHILE
        Clear OUTPUT_BUFFER
        Increment column in DIGRAM_MATRIX
      END_WHILE
      Increment row in DIGRAM_MATRIX
    END_WHILE
    Close output file
  ELSE { output file open unsuccessful }
    Display error message regarding unsuccessful file open
  END_IF


{ ADD WORDS TO DICTIONARY FILE }
ELSE_IF (MODE is STORE_WORDS) THEN
  IF (no filenames in FILE_LIST) THEN
    Prompt user for file(s) to be processed
    IF (filename(s) inputted by user) THEN
```

```
      Store filename(s) into record(s) in FILE_LIST
    ELSE { no filename(s) inputted by user }
      END_PROGRAM
    END_IF
  END_IF
  WHILE (more files to process)
    Open current input file for reading only
    IF (input file open successful) THEN
      WHILE (not end of current input file)
        Read next record from input file into INPUT_BUFFER
        WHILE (not end of CURRENT_RECORD)
          Update CURRENT_WORD_BEGIN pointer
          Check current char against WORD_DELIM
          If (current char is not delimiter) THEN
            WHILE (delimiter not encountered)
              Increment current char
            END_WHILE
            Update CURRENT_WORD_END pointer
          ELSE { current char is delimiter }
            WHILE (current char is delimiter)
              Copy char to OUTPUT_BUFFER
              Increment current char
            END_WHILE
          END_IF
          Copy word to be checked into CURRENT_WORD
          Convert CURRENT_WORD to uppercase
          Check CURRENT_WORD against LEGAL_DIGRAMS
```

```
          IF (all digrams in CURRENT_WORD are legal) THEN
            Check CURRENT_WORD against COMMON_WORDS
            IF (CURRENT_WORD not found in COMMON_WORDS) THEN
              Use DIGRAM_MATRIX to index into DICTIONARY
              Search DICTIONARY for CURRENT_WORD
              IF (CURRENT_WORD not found in DICTIONARY) THEN
                WHILE (CURRENT_WORD letter not in DICTIONARY)
                  Add CURRENT_WORD letter to DICTIONARY
                END_WHILE
              END_IF
            END_IF
          ELSE { not all digrams in CURRENT_WORD are legal }
            Inform user of incorrect word in input file
          END_IF
        END_WHILE
      END_WHILE
      Close input file
    ELSE { input file open unsuccessful }
      Display error message regarding unsuccessful file open
    END_IF
    Get next filename from FILE_LIST
  END_WHILE


{ CREATE NEW MAIN DICTIONARY FILE }
ELSE_IF (MODE is NEW_DICT) THEN
  IF (no filenames in FILE_LIST) THEN
```

```
    Prompt user for file(s) to be processed
    IF (filename(s) inputted by user) THEN
      Store filename(s) into record(s) in FILE_LIST
    ELSE { no filename(s) inputted by user }
      END_PROGRAM
    END_IF
  END_IF
  Create/open new file for DICTIONARY ARRAY
  IF (new file create/open successful) THEN
    WHILE (more files to process)
      Open current input file for reading only
      IF (input file open successful) THEN
        WHILE (not end of current input file)
          Read next record from input file into INPUT_BUFFER
          WHILE (not end of CURRENT_RECORD)
            Update CURRENT_WORD_BEGIN pointer
            Check current char against WORD_DELIMS
            If (current char is not delimiter) THEN
              WHILE (delimiter not encountered)
                Increment current char
              END_WHILE
              Update CURRENT_WORD_END pointer
            ELSE { current char is delimiter }
              WHILE (current char is delimiter)
                Copy char to OUTPUT_BUFFER
                Increment current char
              END_WHILE
```

```
            END_IF
            Copy word to be checked into CURRENT_WORD
            Convert CURRENT_WORD to uppercase
            Check CURRENT_WORD against LEGAL_DIGRAMS
            IF (all digrams in CURRENT_WORD are legal) THEN
              Check CURRENT_WORD against COMMON_WORDS
              IF (CURRENT_WORD not found in COMMON_WORDS) THEN
                Use DIGRAM_MATRIX to index into DICTIONARY
                WHILE (CURRENT_WORD letter not in DICTIONARY)
                  Add CURRENT_WORD letter to DICTIONARY
                END_WHILE
              END_IF
            ELSE { not all digrams in CURRENT_WORD are legal }
              Inform user of incorrect word in input file
            END_IF
          END_WHILE
        END_WHILE
        Close input file
      ELSE { input file open unsuccessful }
        Display error message regarding unsuccessful file open
      END_IF
    Get next filename from FILE_LIST
    END_WHILE
  ELSE { new file create/open unsuccessful }
    Display message regarding unsuccessful new file create/open
  END_IF
```

```
{ CHANGE WORDS IN DICTIONARY FILE }
ELSE_IF (MODE is MODIFY_WORDS) THEN

  *** AT PRESENT, THIS MODE WILL NOT BE INCLUDED AS ONE OF THE
  *** QUALIFIERS FOR THE SPELLER --- WORD MODIFICATION WILL BE
  *** ACCOMPLISHED BY LISTING OUT THE CONTENTS OF THE DICTIONARY
  *** FILE, EDITING THE INCORRECT WORDS IN THIS LIST, AND THEN
  *** CREATING A NEW DICTIONARY FILE FROM THE UPDATED LIST.


{ REMOVE WORDS FROM DICTIONARY FILE }
ELSE_IF (MODE is DELETE_WORDS) THEN

  *** AT PRESENT, THIS MODE WILL NOT BE INCLUDED AS ONE OF THE
  *** QUALIFIERS FOR THE SPELLER --- WORD DELETION WILL BE
  *** ACCOMPLISHED BY LISTING OUT THE CONTENTS OF THE DICTIONARY
  *** FILE, DELETING THE INCORRECT WORDS FROM THE LIST, AND THEN
  *** CREATING A NEW DICTIONARY FILE FROM THE UPDATED LIST.


{ DISPLAY HELP INFORMATION }
ELSE_IF (MODE is HELP) THEN
  Open HELP_INFO file
  IF (HELP_INFO file found) THEN
    Open HELP_INFO file
    Display contents of HELP_INFO file to user
  ELSE { HELP_INFO file not found }
    Display message as to unavailability of HELP_INFO file
  END_IF
```

```
END_IF

END_PROGRAM
```

5 INTERFACE SPECIFICATION

5.1 External Interfaces

- Input File

Depending on the argument passed on the SPLCHK command line, the input to the program can be in one of several forms:

(1) A user-created file, preferably of sequential structure and containing text, which will be checked for spelling errors by the SPLCHK program.

(2) User input from the keyboard, for example as input to the word checking mode of SPLCHK.

(3) A sequential structure text file of words to be added to the SPLCHK dictionary file.

- Output File

Created by the SPLCHK program, the output file is a sequentially structured text file which contain(s) the input file data, along with user-accepted substitutes for input file misspellings. There is a one-to-one correspondence between an input file and an output file; that is, there will be one output file created for each file that a user submits to the SPLCHK program.

- System Files

(1) Master Dictionary file - (SPLCHK.DCT) - created, maintained, and accessed by the SPLCHK program, the "dictionary" is a direct-access stream file containing the trie-structure representation of the dictionary used by the SPLCHK program.

The record structure for this file (see Figure 2 at the end of this section) is as follows: a quadword (eight-byte) record, in which the first eight bits represent a particular character, the next twenty-eight bits are a child pointer, and the last twenty-eight bits are a sibling pointer. If the character in the first field is the character being searched for, then the child pointer may be followed down to the next level of the trie. If the character is not the one being searched for, however, the sibling pointer can be followed to other letters at the same level of the trie. The first twenty-six records in the file will be the starting records for each of the twenty-six letters of the alphabet (i.e. they will each contain a letter that follows the particular letter of the alphabet in a word stored in the dictionary).

More details on the fields in a dictionary record are given in Section 6.2 (Internal Interfaces).

Two methods for the implementation of a trie structure on disk were considered and evaluated: (1) the use of a linked list, and (2) the use of a sparse matrix. The linked list approach was chosen because of the savings in terms of disk space that it provided over the other approach.

(2) Informational file - (SPLCHK.INF) - this file contains the latest indices into the Master Dictionary file (next empty position, digram matrix), as well as the latest values for the unigrams, digrams, trigrams, and digram words arrays. It is updated each time that words are added to the SPLCHK dictionary. If this file becomes corrupted or deleted, then the SPLCHK program cannot run properly, as it cannot correctly access the words stored in the Dictionary file.

5.2 Internal Interfaces

This section describes the physical layout and implementation of the various interfaces and structures that are used in the SPLCHK program. Information as to the higher-level uses of these items is contained in two later sections of this document: Section 7.2, Data Bases, and Section 7.3, Communication Among Modules.

The following structures are used in the SPLCHK program:

(A) COMMON

ARRAY OF THE 300 MOST COMMON WORDS - a static array of 4-byte records consisting of:

bit 0	word terminator indicator
bits 1-7	character
bits 8-19	child/next pointer
bits 20-31	sibling/alternate pointer

See Figure 2 at the end of this section.

(B) DICT

BUFFER USED TO CONTAIN CURRENT DICTIONARY RECORD - the current Master Dictionary record is read into this dynamic buffer, which contains the following fields:

bit 0	word terminator indicator
bits 1-7	character
bits 8-35	child/next pointer
bits 36-63	sibling/alternate pointer

See Figure 2 at the end of this section.

(C) DIGRAM_MATRIX

ENTRY POINTS INTO DICTIONARY FOR TWO-LETTER/NUMBER COMBINATIONS - this two-dimensional static array (indexed in either direction by 0-9,A-Z) will contain the record

number entry points into the main dictionary array/file; the first two characters in a word will be used as indices into this array, and the array element located will be the character in the dictionary file which will be compared against the third character in the word being checked.

(D) DIGRAM_WORDS

ARRAY OF TWO-LETTER WORDS - a dynamic bit array indicating whether or not a particular two-letter combination is a correctly spelled word (i.e. at, is). A bit turned on indicates TRUE; a bit turned off indicates FALSE.The array is accessed by the formula:

[(ASCII(letter_1 - 1) * 26) + ASCII(letter_2)]

(E) DIGRAMS

ARRAY OF DIGRAMS - a dynamic bit array indicating whether or not a particular two-letter combination is contained in a correctly spelled dictionary word (i.e. "ab" in "about"). A bit turned on indicates TRUE; a bit turned off indicates FALSE.

(F) GUESSES

LINKED LIST OF "CORRECT SPELLING" GUESSES - a dynamic linked list of records containing: correct spelling guesses (character strings) and pointers to the next guess in the list.

(G) LINE_BUFFER

INPUT DATA BUFFER - a dynamic array of 5 records consisting of: a number of characters, and a 255 position character string; the 5 records are, respectively:

- 2 lines before the line currently being checked
- the middle line, currently being checked for spelling (LINE_BUF)
- 2 lines after the line currently being checked

WORD_BUF is a structure of type LINE_BUFFER that contains the word from LINE_BUF which is currently being checked for spelling correctness.

(H) LINE_PTR

DISPLAY "BUFFER" POINTERS FOR INCORRECT WORD "IN CONTEXT" - a dynamic array of 5 pointers - one for each of the 5 character strings in the LINE_BUFFER; when a new record from the file is read into the LINE_BUFFER, the pointers are updated to reflect the new positions of the five "LINE BUFFER" records.

(I) NEXT_EMPTY

NEXT AVAILABLE POSITION IN MASTER DICTIONARY - dynamic unsigned value which indicates the next available position in the Master Dictionary file.

(J) OUTPUT_BUFFER

OUTPUT DATA BUFFER - a dynamic record consisting of: a number of characters, and a 255 position character string.

(K) SUBS_PAIRS

SUBSTITUTION PAIRS TABLE - a dynamic array of records containing: for a word to be substituted for, the length of the word and the characters it contains; and for the word to be used as the substitute, the length of the word and the characters it contains.

(L) TRIGRAMS

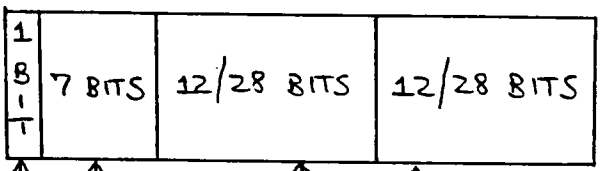
ARRAY OF TRIGRAMS - a dynamic bit array indicating whether or not a particular three-letter combination is contained in a correctly spelled dictionary word (i.e. "abo" in "about"). A bit turned on indicates TRUE; a bit turned off indicates FALSE.

(M) UNIGRAMS

ARRAY OF UNIGRAMS - a dynamic bit array indicating whether or not a particular letter is a correctly spelled word (i.e. a, i). A bit turned on indicates TRUE; a bit turned off indicates FALSE.

(N) WORD_BUF

BUFFER CONTAINING CURRENT UPPERCASE WORD - a dynamic character string containing uppercase version of the word currently being checked for spelling correctness (the same word that is currently pointed to in LINE_BUF by its BEGIN and END pointers).



SIBLING POINTER
CHILD POINTER
CHARACTER
WORD TERMINATOR FLAG

COMMON WORDS – 12 BIT POINTERS
32 BITS PER RECORD

DICTIONARY – 28 BIT POINTERS
64 BITS PER RECORD

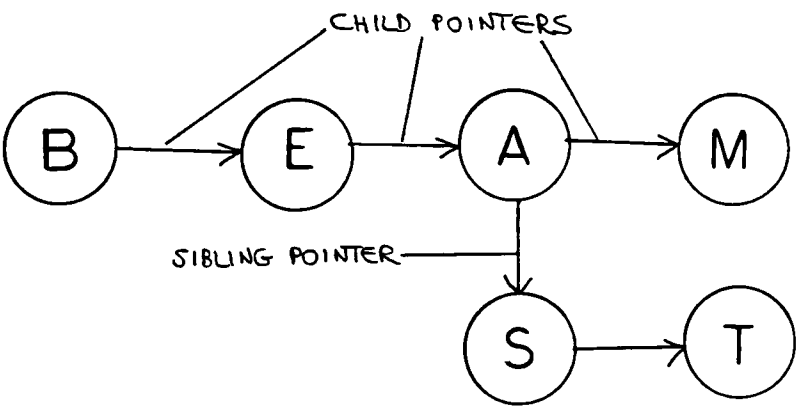


Figure 2

6 MODULE DESIGNS

6.1 Processor Modules

NOTE: An * by a module name indicates a utility routine, used as support for, but not as a part of, the SPLCHK program.

(1) ADD_A_WORD

This routines adds a new word to the SPLCHK dictionary file. It will update all necessary dictionary records, as well as the DIGRAM_MATRIX, if necessary.

(2) ADD_DIGRAM

This routine turns on the appropriate bit in the DIGRAMS matrix for each digram in the routine's parameter.

(3) ADD_DIGRAM_WORD

This routine turns on the appropriate bit in the DIGRAM_WORDS array for the digram word contained in the routine's parameter.

(4) ADD_MATRIX

Updates an element in the DIGRAM_MATRIX to reflect a newly-added dictionary entry.

(5) ADD_TRIGRAM

This routine sets a bit in the TRIGRAMS matrix for each of the trigrams in the TEXT field of the routine's argument.

(6) ADD_UNIGRAM

This routine sets a bit in the UNIGRAMS array for the parameter "unigram."

(7) ADD_WORDS

This routine opens a user-specified input file, and then opens the SPLCHK dictionary file for NEW or ADD access, depending on a user-specified MODE. The routine then reads through the input file, checking words against both the COMMON WORDS list and the dictionary, and adds to the dictionary any words not found. It also updates the UNIGRAMS, DIGRAMS, TRIGRAMS, and DIGRAM_WORDS data structures as appropriate.

(8) CHECK_MATRIX

Checks the Digram Matrix for a particular two-letter combination, and returns the starting location for that combination in the dictionary file.

(9) CHECK_SUBSTITUTION_PAIRS

Checks the SUBS_PAIRS linked list for a previous occurrence of the same misspelled word, and returns a pointer to the misspelled/correct pair, if found.

(10) CHECK_DIGRAMS

Checks whether digram combinations encountered in the current word are contained in any dictionary words.

(11) CHECK_DIGRAM_WORD

Checks whether a two-letter combination is a correctly-spelled word by searching for it in the DIGRAM_WORDS array.

(12) CHECK_TRIGRAMS

Checks whether trigram combinations encountered in the current word are contained in any dictionary words.

(13) CHECK_UNIGRAM

Checks whether or not a one-character word is in the UNIGRAMS array.

(14) CLEAR_MATRIX

Clears and initializes the Digram Matrix.

(15) CONTENTS

Lists the words stored in the dictionary file SPLCHK.DCT.

(16) DISPLAY_CONTEXT

Displays a misspelled word "in context;" i.e., in its original line of text, which is surrounded by the

preceeding two and following two lines of text.

(17) EXCLUDE

Excludes characters which are not uppercase letters. Used mainly to avoid using the numeric characters 0-9 when generating spelling guesses.

(18) TRY_EXTRA_LETTER

Generates possible correct spellings for a word by removing a letter from each position in the word.

(19) FILE_CHK

Checks the spelling of words in a user file, and reports incorrectly-spelled words.

(20) FILE_CORRECT

Checks and corrects (through user interaction) the spelling of words in a file.

(21) FREE_GUESSES

Frees the space allocated for the guess list.

(22) GEN_CMN *

Generates the data for the Common Words list, based on an input file containing the Common Words.

(23) GEN_GUESSES

Calls the four SPLCHK guess generation routines (for missing letter, wrong letter, extra letter, and transposed letters) which generate possible correct spellings for a misspelled word.

(24) GET_SELECTION

Accepts user correction selection, checks for validity, and returns selection number, along with selected correction in parameter NEW_WORD.

(25) HELP

Displays SPLCHK help information, regarding calling formats, arguments, etc.

(26) INIT_GUESSES

Initializes GUESSES linked list.

(27) INIT_SUBS_PAIRS

Initializes linked list of substitution pairs.

(28) LIST_CMN *

Lists the contents of the Common Words list,except for one- and two-letter words.

(29) LIST_GUESSES

This routine displays all the correct spelling guesses for an incorrect word.

(30) LIST_DIGRAMS *

Creates and prints the DIGRAMS and DIGRAM_WORDS arrays for the Common Words list.

(31) TRY_MISSING_LETTER

Generates possible correct spellings for a word by substituting in turn each letter of the alphabet in between each letter position in the word, as well as at the beginning and at the end of the word.

(32) GET_NEXT_WORD

Gets the next word to be checked from the current input buffer, and updates associated pointers. Also signals end of input file, and copies delimiters from input buffer to (temporary) output buffer.

(33) OFFSET

Returns the offset of a character into the string of 36 (0-9,A-Z) characters used as word components in SPLCHK (i.e. offset(0) = 0, offset(9) = 9, offset(a) = 10, offset(z) = 35).

(34) OPEN_FILE

Opens a user-specified file in a user-specified mode, and returns a pointer to that file. In case of error, prints a message relating to the file in question, and returns NULL.

(35) PARSE

Parses the command line argument to the SPLCHK program. Check for correct number and format of arguments. Returns NULL if a valid mode is not found; otherwise, returns the letter of the mode found.

(36) PRINT_DIGRAMS *

Prints the contents of the DIGRAMS array, as well as the legal digram combinations, indicating which digrams occur in one or more dictionary words.

(37) PRINT_DIGRAM_WORDS *

Prints the current contents of the DIGRAM_WORDS array, as well as the legal digram words.

(38) PRINT_TRIGRAMS *

Prints the current contents of the TRIGRAMS array, indicating which trigram combinations occur in one or more dictionary words.

(39) READ_INFO

Reads the latest values of the UNIGRAMS, DIGRAMS, TRIGRAMS, NEXT_EMPTY, DIGRAM_WORDS, and DIGRAM_MATRIX structures from the SPLCHK.INF file, and stores them into the appropriate data structures.

(40) REMOVE_LWS

Removes leading white space from a buffer containing a word to be checked.

(41) SEARCH

Searches various "dictionary" structures for the current word.

(42) SEARCH_CMN

Searches Common Words list.

(43) SEARCH_DCT

Searches the Dictionary file for the input word.

(44) SPLCHK

SPLCHK main program - this module invokes a particular function of the spelling checker, based on a user-entered command line parameter.

(45) ADD_SUBSTITUTION_PAIR

Allocates a SUBS_PAIRS linked list element, and stores the latest combination of misspelled/correct words into this list element.

(46) STORE_GUESS

This routines allocates a linked list element, into which is stored the latest correct spelling guess.

(47) TIMER *

Allows a section of code to be timed, by calling this routine at the beginning of the section with a state of 0, and at the end of the section with a state of 1.

(48) TRY_TRANSPOSED_LETTERS

Generates possible correct spellings, by transposing each pair of letters in a word.

(49) TRIGRAMS

Checks all combinations of trigrams in an input word against the TRIGRAMS array used by the SPLCHK program.

(50) UPDATE_ERR_BUF

Fills in the error display line, which gets printed below the line with the misspelled word, and which indicates the misspelled word with caret ("^") characters.

(51) UPDATE_LINE_BUFFER

Updates the five line pointers which contain the display "context" area for SPLCHK. At end of file, it NULL's the pointer, and places a NULL in the line buffer. The routine reads two lines ahead of the currently-checked line, so that it can display five lines of "context" when necessary.

(52) UPDATE_OUT_BUFFER

Updates the output buffer with the user-selected substitute for a misspelled word.

(53) WORD_CHK

Searches the Dictionary file (SPLCHK.DCT) for the correct spelling of words input by the user. If the spelling is found to be correct, an affirmative message is printed; otherwise, guesses for a correct spelling are generated and displayed for the user.

(54) WRITE_INFO

Writes the latest values of the UNIGRAMS, DIGRAMS, TRIGRAMS, NEXT_EMPTY, DIGRAM_WORDS, and DIGRAM_MATRIX structures to the informational file SPLCHK.INF.

(55) TRY_WRONG_LETTER

Generates possible correct spellings by substituting each letter of the alphabet, in turn, in each letter position in the original word.

6.2 Data Bases

The SPLCHK program contains several internal databases, which are used to store various types of information, and which are accessed throughout the program.

The physical layout of the items in these databases is described in the Internal Interfaces section of this document (section 6.2). This section describes the higher-level layout of the databases, and their use with relation to the SPLCHK program.

The following are the databases used in the SPLCHK program:

(A) COMMON

This is the Common Words List, which contains the three hundred most common English language words according to the Brown Corpus [10]. The format of this database is an array of 800 elements, each of which is 32 bits long and contains a character, a word terminator indicator, a sibling pointer, and a child pointer.

Words are generated from this database by indexing into one of the first twenty-six elements of the array based on the first character of the word in question, and then following either sibling or child pointers to retrieve the rest of the characters which make up that word.

This database is hard-coded into the SPLCHK program.

(B) DIGRAMS

This database contains information as to whether or not particular digram combinations have been used by one or more dictionary words. The data structure is a bit vector with a length of 1296 bits, which is indexed by a value containing: the sequence number of the first character in the digram multiplied by thirty-six, added to the sequence number of the second character in the digram.

The bits for all digrams for dictionary words are turned on at dictionary update time. Therefore, if the bit for a particular digram is not set, then the dictionary need not be searched for any word containing that digram.

This database is stored in the informational file SPLCHK.INF, and is read from that file at program startup.

(C) DIGRAM_MATRIX

This database contains the offsets into the dictionary file for all of the currently-used digram combinations. The data structure used is a thirty-six by thirty-six matrix of longwords, which is indexed by the two characters of a particular digram.

Whenever a new digram bit is set in the DIGRAMS vector, the location in DIGRAM_MATRIX corresponding to that digram is updated with the number of the next available slot in the dictionary file (this information is stored in the global variable NEXT_EMPTY).

The dictionary file entries actually begin with the third character of the word being stored; the first two character being checked against the DIGRAM_MATRIX database, rather than having to access the dictionary file.

The DIGRAM_MATRIX database is stored in the informational file SPLCHK.INF, and is read from that file at program startup.

(D) DIGRAM_WORDS

This database is similar in structure and accessibility to the DIGRAMS database, except that it contains digram combinations which are correctly spelled words.

As is the case with the DIGRAMS structure, appropriate bits in the DIGRAM_WORDS vector are turned on at dictionary update time for each two-character word that is to be added to the dictionary - the digrams themselves are not actually added to the dictionary file! This means that file access can be avoided when checking the spelling of two-character words.

This database is stored in the informational file SPLCHK.INF, and is read from that file at program startup.

(E) TRIGRAMS

This database contains information as to whether or not particular trigram combinations have been used by one or more dictionary words. It is the similar in structure to the DIGRAMS database, except that it has a length of 46,656 bits. The vector is indexed by a value containing: the sequence number of the first character of the trigram multiplied by thirty-six squared, added to the sequence number of the second character of the trigram multiplied by thirty-six, added to the sequence number of the third character of the trigram.

Again, as with digrams, the bits for the trigrams for dictionary words are turned on at dictionary update time. Therefore, if the bit for a particular trigram is not set, then the dictionary need not be searched for any word containing that trigram.

The TRIGRAMS structure is stored in the informational file SPLCHK.INF, and is read from that file at program startup.

(F) UNIGRAMS

This database is similar in structure and accessibility to the DIGRAMS database, except that it contains information as to which unigrams are correctly spelled words. The UNIGRAMS database is a vector of 36 bits, which is indexed by the sequence number of the unigram character.

As in the cases of several of the other structures used in SPLCHK, an appropriate bit in the UNIGRAMS vector is turned on at dictionary update time for each one-character word that is to be added to the dictionary. Again, as per the digrams, the unigrams themselves are not actually added to the dictionary file, and file access can be avoided when checking the spelling of one-character words.

This database is also stored in the informational file SPLCHK.INF, and is read from that file at program startup.

6.3 Communications Among Modules

The SPLCHK program's modules communicate with each other

through the use of several globally-accessible variables and structures. These items contain various forms of information, and are accessed in different portions of the program.

The physical layout of these items is described in detail in the Internal Interfaces section of this document (section 6.2). In this section, the actual use of the structures with respect to inter-module communication is examined.

The following items are used for communication within the SPLCHK program:

(A) GUESSES

This linked list contains all of the "correct spelling" guesses generated for a particular incorrect word.

This list is maintained by means of head and tail pointers (indicating the beginning of the list and the last-filled slot in the list, respectively), and is "emptied out" after each set of guesses has been displayed and acted on by the user.

(B) LINE_BUFFER

This structure is a queue of five lines (and their associated length values), which contain the current "context" (the two previous lines, and the two following lines) for the current "line" and "word." This array

represents a "scrolling region" into the file which is being checked for spelling. Each latest-read line is added to the bottom of the queue, while the earliest-read line falls off the top of the queue.

To ease access to and manipulation of this structure in various routines throughout the SPLCHK program, each line of LINE_BUFFER is pointed to by an associated pointer in the array LINE_PTR.

(C) NEXT_EMPTY

This is an item which contains the number of the next available slot in the dictionary file. The slot numbers are numbered sequentially, beginning at 0, and are multiplied by the size of a dictionary item (64 bits or 2 longwords) to yield the correct offset into the file.

NEXT_EMPTY is stored in the informational file SPLCHK.INF, and is read at program startup.

(D) OUTPUT_BUFFER

This item consists of a text string and an associated length field, and is used to hold the current output line during file correction mode. The OUTPUT_BUFFER is filled as the words in the current input line are checked for correct spelling. When the processing of the input line has been completed and all words have been moved into the OUTPUT_BUFFER, the text string portion of this item is

then written to the output file.

(E) SUBS_PAIRS

This linked list contains pairs of words: the first, a previously-found, incorrectly-spelled word; the second, a user-selected replacement for the misspelled word. Also contained in the data structure are the associated lengths of the two words.

This list is maintained by the use of head and tail pointers (which, as was the case with GUESSES, indicate the beginning of the list and the last-filled slot in the list, respectively), and is updated each time a new misspelled word is discovered and corrected.

Whenever a misspelled word is found by SPLCHK, the first words of the SUBS_PAIRS elements are searched for the incorrect word. If found, the second word of that linked list element is offered to the user as a "correct" guess.

7 VERIFICATION AND VALIDATION

7.1 Test Plan

The test plan for the SPLCHK spelling checker and corrector was designed to check various phases of the software, including functionality, consistency, correctness, speed, error handling, and to a small degree, user-friendliness and ease of use. The plan includes the following criteria for testing the SPLCHK program:

(A) Verifying the different functions of the SPLCHK program, based upon command line parameters, as described in the user documentation.

(B) Assuring that similar functions are handled in a fairly consistent manner: parameters, prompting, inputs and outputs, help information, guesses for words in different modes, and user displays.

(C) Comparing the functionality and outputs of the SPLCHK program to various other spelling checkers and correctors that operate in the VAX/VMS environment.

(D) Validating the accuracy of the spelling checking and spelling correction portions of the SPLCHK program, based on known inputs: correctly and incorrectly spelled words, words in the dictionary, words not in the dictionary, words with non-alphanumeric characters, and so on.

(E) Validating the correct addition of words to the SPLCHK dictionary in "new dictionary" and "dictionary update" modes, again based on known inputs: a known list of words, the contents of which can later be checked against the SPLCHK dictionary.

(F) Testing the SPLCHK program under various types of error conditions: incorrect parameters or data, "bad" data, various files missing or inaccessible, words containing special characters, etc.

7.2 Test Procedures

The test procedures involved the implementation of some of the test plan criteria in the form of actual inputs to the SPLCHK program. Among the procedures used to test the program were the following:

(A) Trying the different SPLCHK command line parameters, as well as incorrect parameters and no parameters at all, and verifying that the results matched those specified in the user documentation.

(B) Checking that the command line arguments are all entered in a similar manner, that prompts are displayed consistently, that inputs and outputs are handled in similar fashion throughout the various functions, that help information is consistent with the user documentation, and that words are checked

consistently by different modes of the program.

(C) Comparing the results in like functions of the SPLCHK program versus the DECUS SPELL program (file and word check modes) and DECspell (file correction mode).

(D) Testing the different checking and correcting modes of the SPLCHK program with known correct and incorrect sets of inputs, and verifying the results.

(E) Adding known sets of words to the SPLCHK dictionary in both "new dictionary" and "dictionary update" modes, and verifying their correct placement in the dictionary by using the "word check" or "list contents" modes.

(F) Testing the SPLCHK program with incorrect words, words not in the dictionary, and words containing non-alphanumerics; and renaming the dictionary and informational files, to make them inaccessible.

7.3 Test Results

The following are the results of performing the tests described in the previous section:

(A) Testing of the parameters and functionality of the spelling checker was successful; the various modes were all accessible according to the specified parameters, and incorrect or non-existent parameters

produced an error message and help text.

(B) The testing for consistency in various phases of the program also proved successful; parameters, help text, and documentation are consistent, inputs and outputs in the various modes of the program are handled in a consistent manner (e.g. a similar input can be entered for similar functions, and will be handled in a similar manner), and words checked in the different modes of the program are consistently found correct or incorrect throughout.

(C) The spelling checker comparison test was not entirely successful; in the first portion of the test, and because the SPLCHK dictionary is based on the DECUS SPELL program's dictionary, the misspelled words found by the two programs were the same, and the SPLCHK program was not appreciably faster or slower than the DECUS SPELL program during the checking portion, although the startup of the DECUS SPELL program was noticeably slower.

In the second portion of the test, the DECspell program was much faster than the SPLCHK program, both in generating guesses and in correcting the file, and also seemed much better equipped to handle correction of misspelled words, in terms of available options. DECspell was a bit slower during startup than was

SPLCHK, but this seemed to be due to it locating an entire page or more of spelling errors before displaying them. Even though SPLCHK did generate more guesses for words in many cases than did DECspell, a great amount of time was taken to generate those guesses.

(D) In the known inputs test, the SPLCHK program was successful; words known to be in the dictionary were indicated as being correct, while words known not to be in the dictionary were signalled as being misspelled.

(E) The SPLCHK program was also successful in the known dictionary additions test: known words were added to both existing and newly created dictionaries, and were then verified using the "word check" or "list contents" functions of the program.

(F) The test of various error conditions was not entirely successful. The SPLCHK program handled inaccessible files with appropriate error messages, and reacted appropriately to bad or missing parameters, but had problems consistently handling bad inputs in its different modes. Whereas correct and incorrect alphanumeric words seemed to be handled consistently throughout the program, words containing special characters were not.

8 CONCLUSIONS

8.1 Problems Encountered And Solved

During the process of design and implementation of the SPLCHK program, several problems were encountered which had to be resolved before the project could proceed.

One of the first problems to be dealt with was the decision on a method of storing on disk a trie structured dictionary. A primary goal of this project was to evaluate whether or not a trie structure was a good method for implementing a large spelling checker dictionary, and whether or not this type of structure would be efficient in terms of size and access speed.

The decision was made to store the dictionary trie on a node per file record basis, with each node consisting of a character, a word terminator flag, and sibling and child pointers. This structure was used for both the internal Common Words list and the actual dictionary file out on disk, with the major difference being in the sizes of the sibling and child pointer fields (the pointer fields were 28 bits long for the dictionary file, while the Common Words list pointer fields were only 12 bits long).

Refer to sections 9.2 and 9.3.1 for more discussion of this topic.

Another area of concern to be addressed was the generating of possible replacement candidates for words not located in the speller's dictionary. A side issue here was an attempt to narrow down as much as possible the list of replacement candidates, to avoid more dictionary access than absolutely necessary.

In the SPLCHK program, guesses as to correct spellings are generated from permutations of the "incorrect" word, using an algorithm based on Damerau's findings [4]. These guesses are then checked in the same manner as input words.

The guesses are first checked against the Digrams and Trigrams arrays. If any of the digrams or trigrams in the "guess" are not found in the respective arrays, the word is known not to be in the dictionary.

Secondly, the guesses are checked against the Common Words list. Since the Common Words list holds 300 of the most common words in the English language, this checking should catch a large percentage of words in most documents.

If the guess is not located in the Common Words list, then finally the dictionary file is searched for the word.

Only those guesses found in either the Common Words list or the dictionary are actually displayed to the user as possible correct replacements.

Refer to sections 9.2 and 9.3.1 for more discussion of this topic.

The problem of data corruption was also a concern during the design and implementation stages. Specifically, this concern deals with the corruption of the SPLCHK dictionary and informational files, during the "new dictionary" and "dictionary update" phases. Since the dictionary file contains the main structure of words in the dictionary, and the informational file contains indices into the dictionary file as well as data used to initialize various SPLCHK structures, corruption of either of these files would render the SPLCHK program useless.

Data corruption in these files occurs when either the "new dictionary" or "dictionary update" phase is interrupted or terminated prematurely.

The current workaround for possible data corruption is to create backup copies of the dictionary and informational files before either updating the current dictionary or creating a new one. Therefore, if an error does occur during dictionary update, the new (corrupted) files can be deleted, and copies of the backup files can be used to redo the update process.

Refer to sections 9.2 and 9.3.1 for more discussion of this topic.

Addition of words to the dictionary file was fully implemented in this version of SPLCHK. However, deletion and modification of dictionary words is not currently implemented per se, and requires a fairly roundabout process to achieve.

The dictionary contents must be listed to an output file, the contents of that file must be deleted and/or modified, then a new dictionary file must be created and updated with the modified list of words. This is a cumbersome and time-consuming process, and probably the type of task to be performed overnight or during a weekend.

Refer to sections 9.2 and 9.3.1 for more discussion of this topic.

Yet another problem which had to be solved was the handling of special characters in the spelling checker input, specifically characters other than alphanumerics. These special characters are currently treated as delimiters: if a non-alphanumeric character is encountered in the speller's input, it is considered a word delimiter, and the alphanumeric text preceding the "delimiter" is then taken as the word to be checked against the dictionary.

Alphanumerics themselves were also a issue that had to be resolved during the design of this program. Alphanumerics are handled by using "sequence numbers" (see Glossary) to allow both alphabetic and numeric information in the majority of SPLCHK's data structures. Therefore, words containing

alphanumeric characters can be stored in the SPLCHK dictionary, and can be verified against the dictionary.

However, generation of correct spelling guesses does not include numeric characters, in order to avoid the overhead of ten more searches for most permutations of the misspelled word. For the relatively small number of alphanumeric words that would be generated, this was felt to be a reasonable tradeoff.

Another problem area is that the correlation of mixed-case input and output is currently handled only in some cases. Any text copied from the input file to the output file retains its original case. User entered replacements for incorrect words also retain their original case. However, guesses generated by SPLCHK are currently lowercase only.

Refer to sections 9.2 and 9.3.1 for more discussion of this topic.

8.2 Discrepancies And Shortcomings Of The System

The SPLCHK program, unfortunately, has several shortcomings, as well as several unimplemented or partially-implemented items.

Among the more important shortcomings is the fact that the dictionary file is very large for the amount of words contained in it, and that access to it is quite slow, most likely because of the size and the number of accesses required

alphanumeric characters can be stored in the SPLCHK dictionary, and can be verified against the dictionary.

However, generation of correct spelling guesses does not include numeric characters, in order to avoid the overhead of ten more searches for most permutations of the misspelled word. For the relatively small number of alphanumeric words that would be generated, this was felt to be a reasonable tradeoff.

Another problem area is that the correlation of mixed-case input and output is currently handled only in some cases. Any text copied from the input file to the output file retains its original case. User entered replacements for incorrect words also retain their original case. However, guesses generated by SPLCHK are currently lowercase only.

Refer to sections 9.2 and 9.3.1 for more discussion of this topic.

8.2 Discrepancies And Shortcomings Of The System

The SPLCHK program, unfortunately, has several shortcomings, as well as several unimplemented or partially-implemented items.

Among the more important shortcomings is the fact that the dictionary file is very large for the amount of words contained in it, and that access to it is quite slow, most likely because of the size and the number of accesses required

to retrieve an entire word.

The main reason for the large file size is the fact that each node in the dictionary takes up eight bytes of space, mostly for pointers. The amount of disk space saved by the elimination of redundancy of common word roots seems to be more than negated by the amount of space necessary to store the pointers to the nodes of the trie.

A cause of the slow "word" retrieval of the SPLCHK program is the number of file accesses that must be made in order to retrieve an entire word, especially if many sibling and child pointers must be checked before the correct ones are found. This is a function of the number of words in the dictionary having a common root: redundancy of the characters of the root is eliminated, but more child and or sibling pointers have to be checked to proceed in the trie.

Another shortcoming displayed by the program is that the case of guesses generated by SPLCHK is not correlated to the case of the input word. This leads to output files which are not case-consistent with the input files upon which they are based, and may have to be edited after the spelling correction process.

Possible corruption of dictionary and informational files, as described in the previous section, is yet another shortcoming of the SPLCHK program. This problem is currently controlled by the documented workaround, and occurs very

infrequently. Extensive revision of the informational file access routines would be necessary to remedy the problem in the program itself.

Several discrepancies between the original spec and the current SPLCHK program exist. Among these is the non-implementation of a log file for the file correction phase. Because of the interactive nature of file correction, the generation of a log file did not seem to be a necessary or justifiable feature.

Another discrepancy is the non-implementation of a user-specific or secondary dictionary. However, since one of the main goals of this project was to test a trie structure, implementing a second trie structured dictionary seemed redundant, while implementing a different structure for a secondary dictionary seemed to be moving away from the direction of this thesis.

The lack of "real" delete and modify functions in the SPLCHK program is yet another discrepancy of this system; however, it was felt that the deletion and modification process mentioned in the previous section, cumbersome as it is to use, would suffice for this version of the program, as deletion and modification of the dictionary is expected to occur very infrequently.

A final discrepancy of the software is not checking users for "authorized" access. A straightforward method of implementing this was not found; it is left to be added as a future extension.

8.3 Lessons Learned

8.3.1 Alternative Approaches For Improved System

Based on the results of this implementation of the SPLCHK spelling error detector and corrector, several suggestions can be made as to alternative approaches for some of the functionality of the program.

First among these suggestions is the discovery of alternative methods for storing a trie structure on disk, as well as for generating correct spelling guesses. These two areas are by far the weakest in the program.

A different approach to storing a trie structured dictionary would hopefully scale down the size of the dictionary file, and also improve file access to some degree.

A alternate means of generating guesses for correct spellings would hopefully speed up the SPLCHK program with regard to the time involved in displaying guesses for words to the user; for long words, the current time delays are intolerable.

Consideration could also be given to trying the current dictionary structure with a different guessing algorithm, or conversely, the current guessing algorithm with a different dictionary structure.

The use of VAX global sections, or the optimization of some of SPLCHK's input/output routines would be one suggestion for improving the dictionary access times, as well as possibly the numbers of disk accesses overall. Global sections allow the mapping of memory directly to disk, thereby writing directly to disk items that are written to locations in memory. This would not only improve file access, it might also be used in controlling corruption of the dictionary and informational files, since values written into various structures would also be written to disk.

The handling of characters other than alphabetic could also be improved, with regard to consistency in checking and correcting. Numeric characters are currently handled, but at the expense of functions which check for non-alphabetic characters, and proceed in different paths based on the results. Non-alphanumeric characters are currently ignored in most cases, and yet some (such as dashes, apostrophes,etc.) could be handled in various ways.

(C) The implementation of dictionary deletion and modification functionality, which would be a part of the actual software package, and not a workaround implementation (as currently exists).

(D) The inclusion of all "Common Words" in the listing of the SPLCHK dictionary; the words in the "Common" list are currently not shown as part of the dictionary listing.

(E) The checking of user "authorization" when attempting to run the "dictionary update," "new dictionary," or "list contents" phases of the SPLCHK program.

(F) The use of some type of bypass or "ignore" mechanism for text formatting commands, such as those used in RUNOFF, TROFF, NROFF, etc.

(G) The conversion of the SPLCHK software to allow it to operate under the UNIX operating system.

8.3.3 Related Thesis Topics For The Future

The following are some suggestions for thesis topics related to the project described in this document. They are in most cases extensions of some of the research or work done on this project into other, somewhat related, areas.

(A) Other applications for tries; other areas where redundancy can be somewhat eliminated by the use of a trie structure for storing information.

(B) Implementing an optional menu-driven interface for the SPLCHK program, along with various levels of help within the program itself (more assistance for novices, less for experts).

(C) Implementing a spelling checker and corrector with even more of a screen-oriented user interface (much like DECspell); involving some terminal independency, and utilizing various features of different terminals in a consistent manner.

(D) The possibility of performing some type of file compression on a dictionary file such as the one used by SPLCHK: can it be done, what savings would result, what would be the negative aspects, etc.

(E) Including some editing capability within the file correction portion of the SPLCHK program, or interfacing this portion of the program to a text editor.

(F) Making SPLCHK compatible with one or more text editors, to allow a combined use of the two software tools.

9 BIBLIOGRAPHY

(1) Alberga, C.N., "String Similarity and Misspellings," Communications of the ACM, 10, 1967, pp. 302-313.

(2) Bentley, Jon Louis, "Multidimensional Binary Search Trees Used for Associative Searching," Communications of the ACM, 18, 1975, pp. 509-517.

(3) Berger, Kenneth W., The Most Common 100,000 Words Used in Conversations, Kent, Ohio, Herald Publishing House, 1977.

(4) Damerau, Fred J., "A Technique for Computer Detection and Correction of Spelling Errors," Communications of the ACM, 7, 1964, pp. 171-176.

(5) Dodds, D.J., "Reducing Dictionary Size by using a Hashing Technique," Communications of the ACM, 25, 1982, pp. 368-370.

(6) Fredkin, Edward, "Trie Memory," Communications of the ACM, 3, 1960, pp. 490-499.

(7) Jaeschke, G., "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions," Communications of the ACM, 24, 1981, pp. 829-833.

(8) Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Englewood Cliffs, New Jersey, Prentice-Hall, 1978.

(9) Knuth, D.E., "Optimum Binary Search Trees," Acta Informatica, 1, 1971, pp. 14-25.

(10) Kucera, Henry and Francis, W. Nelson, Computational Analysis of Present-Day American English, Providence, Rhode Island, Brown University Press, 1967.

(11) Lemmons, Phil, "Five Spelling-Correction Programs for CP/M-Based Systems," Byte, 1981, pp. 434-448.

(12) Maly, K., "Compressed Tries," Communications of the ACM, 19, 1976, pp. 409-415.

(13) Muth, Frank E., and Tharp, Alan L., "Correcting Human Error in Alphanumeric Terminal Input," Information Processing and Management, 13, 1977, pp. 329-337.

(14) Nix, Robert, "Experience With a Space Efficient Way to Store a Dictionary," Communications of the ACM, 24, 1981, pp. 297-298.

(15) Partridge, D.P., and James, E.B., "Natural Information Processing," International Journal of Man-Machine Studies, 6, 1974, pp. 205-235.

(16) Peterson, James L., "Computer Programs for Detecting and Correcting Spelling Errors," Communications of the ACM, 23, 1980, pp. 676-687.

(17) Peterson, James L., Computer Programs for Spelling Correction, Berlin, Springer-Verlag, 1980.

(18) Robinson, Peter and Singer, Dave, "Another Spelling Correction Program," Communications of the ACM, 24, 1981, pp. 296-297.

(19) Sheil, B.A., "Median Split Trees: A Fast Lookup Technique for Frequently Occuring Keys," Communications of the ACM, 21, 1978, pp.947-958.

(20) Sussenguth, Edward H., Jr., "Use of Tree Structures for Processing Files," Communications of the ACM, 6, 1963, pp. 272-279.

(21) Van Rijsbergen, C.J., Information Retrieval, Boston, Butterworths, 1980, p. 91.

10 APPENDICES

APPENDIX A

COMMON WORDS LIST

The following list of 300 words is derived from the Brown Corpus [10] of the most commonly used words in the English language. The words are shown in alphabetical order, as opposed to order of frequency of occurrence. These words are contained in the "COMMON" hard-coded trie structure of words used in the SPLCHK program.

A	BEGAN	DON'T
ABOUT	BEING	DONE
AFTER	BEST	DOOR
AGAIN	BETTER	DOWN
AGAINST	BETWEEN	DURING
ALL	BIG	EACH
ALMOST	BOTH	EARLY
ALONG	BUSINESS	END
ALSO	BUT	ENOUGH

ALTHOUGH	BY	EVEN
ALWAYS	CALLED	EVER
AMERICAN	CAME	EVERY
AMONG	CAN	EYES
AN	CASE	FACE
AND	CERTAIN	FACT
ANOTHER	CHILDREN	FAMILY
ANY	CHURCH	FAR
ARE	CITY	FELT
AREA	COME	FEW
AROUND	COULD	FIND
AS	COUNTRY	FIRST
ASKED	COURSE	FOR
AT	DAY	FORM
AWAY	DAYS	FOUND
BACK	DEVELOPMENT	FOUR
BE	DID	FROM
BECAUSE	DIDN'T	GENERAL
BECOME	DIFFERENT	GET
BEEN	DO	GIVE
BEFORE	DOES	GIVEN

GO
GOD
GOING
GOOD
GOT
GOVERNMENT
GREAT
GROUP
HAD
HAND
HAS
HAVE
HE
HEAD
HELP
HER
HERE
HIGH
HIM
HIMSELF
HIS
HOME
HOUSE
HOW
HOWEVER
I
IF
LOOKED
MADE
MAKE
MAN
MANY
MATTER
MAY
ME
MEANS
MEMBERS
MEN
MIGHT
MIND
MORE
MOST
MR
MRS
MUCH
MUST
MY
NATIONAL
NEED
NEVER
NEW
NEXT
NIGHT
NO
POSSIBLE
POWER
PRESENT
PRESIDENT
PROBLEM
PROGRAM
PUBLIC
PUT
RATHER
RIGHT
ROOM
SAID
SAME
SAW
SAY
SCHOOL
SECOND
SEE
SEEMED
SENSE
SERVICE
SET
SEVERAL
SHE
SHOULD
SIDE
SINCE

IMPORTANT	NOT	SMALL
IN	NOTHING	SO
INTEREST	NOW	SOCIAL
INTO	NUMBER	SOME
IS	OF	SOMETHING
IT	OFF	STATE
ITS	OFTEN	STATES
JOHN	OLD	STILL
JUST	ON	SUCH
KILL	ONCE	SYSTEM
KIND	ONE	TAKE
KNEW	ONLY	THAN
KNOW	OPEN	THAT
LARGE	OR	THE
LAST	ORDER	THEIR
LATER	OTHER	THEM
LEAST	OTHERS	THEN
LEFT	OUR	THERE
LESS	OUT	THESE
LET	OVER	THEY
LIFE	OWN	THING
LIGHT	PART	THINGS
LIKE	PEOPLE	THINK
LITTLE	PER	THIS
LONG	PLACE	THOSE
LOOK	POINT	THOUGH

THOUGHT	US	WHILE
THREE	USE	WHITE
THROUGH	USED	WHO
THUS	VERY	WHOLE
TIME	WANT	WHY
TO	WAR	WITH
TOLD	WAS	WITHIN
TOO	WATER	WITHOUT
TOOK	WAY	WORK
TOWARDS	WE	WORLD
TURNED	WELL	WOULD
TWO	WENT	YEAR
UNDER	WERE	YEARS
UNITED	WHAT	YET
UNTIL	WHEN	YOU
UP	WHERE	YOUNG
UPON	WHICH	YOUR

APPENDIX B

GLOSSARY

(1) DEC - Digital Equipment Corporation, a manufacturer of computer hardware and software, and specifically the creator and manufacturer of the VAX-11 series of computers and the VMS operating system.

(2) Digram - any combination of two letters (e.g. "ab").

(3) Direct-access File - a file of fixed-length records, in which a record is accessed numerically by its position relative to the beginning of the file (eg. record 1000 is the 1000th record in the file).

(4) Sequence Number - the "positional" numbers assigned to the currently allowable SPLCHK dictionary characters, which are the (lowercase) letters a-z and the digits 0-9. Thirty-six sequence numbers are currently used; they begin at 0 for the character "0" (zero), continue to 9 for the character "9," continue from 10 for the character "a," and end at 35 for the character "z."

(5) Trie - a specialized type of tree structure, used to minimize redundancy. In this application, the letters of a word are stored in the nodes along a path in the trie, and the retrieval of that path will produce the spelling of that particular word. Once letters have been placed in the trie, they can be used in any number of paths of partially similar words; branches in the trie occur where the letters of the words differ. For example, the words "be","bee" and "beet" would require nine characters if stored as separate words in a dictionary. However, using a trie structure, they can be stored in seven characters, since the first two letters, "b" and "e", are common in all three words. Therefore, the seven characters that would need to be stored are "b", "e", a word terminator for "be", a second "e" and a word terminator for "bee", and a "t" and a word terminator for "beet"). The number of characters saved is even greater as the number of words stored is greater, and the redundancy in letters among words increases.

(6) Trigram - any combination of three letters (e.g. "abc").

(7) Unigram - any single letter.

(8) VAX-11 - "Virtual Address Extension," a 32-bit word, supermini series of computers created and manufactured by DEC.

(9) VMS - "Virtual Memory System," the DEC-supplied operating system running on the VAX-11 family of computers.

11 USER MANUAL

SPLCHK

SPLCHK (SPeLling CHecKer) is an interactive spelling checker and corrector currently available for VAX/VMS. Once installed on a system, SPLCHK is accessible by users from DCL command level, and is compatible with VAX-11 sequentially structured files.

SPLCHK currently has seven available functions: (1) a new dictionary file can be created and words inserted in it; (2) words may be added to the current dictionary file; (3) the contents of the dictionary may be listed; (4) user input words can be checked for correctness, with guesses being generated for the incorrectly spelled ones; (5) a file may be checked for correct spelling, with incorrectly spelled words indicated to the user; (6) a file may be checked for correct spelling, with guesses being generated for incorrectly spelled words, and with the ability to change the incorrectly spelled words at that time; and (7) information on using the SPLCHK program may be displayed.

SPLCHK, as mentioned earlier, is accessible from DCL at command level (the "$" prompt) by typing the command "SPLCHK" along with one of the available single character parameters

(described below). Parameters should follow the SPLCHK command, should be separated from it by one or more spaces, and should be preceded by a "/" (slash) or a "-" (dash). Typing the SPLCHK command without any parameters will display a brief "HELP" listing of currently available SPLCHK functions, along with the command format for each function.

The following list shows the command formats for the current SPLCHK functions. The "$" indicates the DCL command prompt, "splchk" is the user entered command, and "-x" indicates a SPLCHK parameter, where "x" can currently be any one of the letters "a" (Add), "c" (list Contents), "f" (check words in File), "h" (Help), "n" (New dictionary), "r" (Replace words in file), or "w" (check Words), depending on the SPLCHK function desired. Input or output files shown in brackets ("[" and "]") are optional; those not shown in brackets are required.

(1) Add new words to the current SPLCHK dictionary:

```
$ splchk -a [inputfile]
```

Inputfile should contain a list of words, one per line, which are to be added to the dictionary. If inputfile is not specified, input will be expected from the terminal keyboard.

(2) List the contents of the current SPLCHK dictionary:

```
$ splchk -c [outputfile]
```

Outputfile will contain the current contents of the dictionary, one word per line. If outputfile is not specified, output will be directed to the terminal display.

(3) List the incorrectly spelled words in a text file:

```
$ splchk -f inputfile [outputfile]
```

Inputfile will be searched for incorrectly spelled words. If inputfile is not specified, an error message will be displayed, and SPLCHK will end processing.
Outputfile will contain the list of incorrectly spelled words, one per line, found in inputfile. If outputfile is not specified, output will be directed to the terminal display.

(4) List SPLCHK help text (the contents of the file SPLCHK.HLP):

```
$ splchk -h [outputfile]
```

Outputfile will contain the SPLCHK "HELP" information. If outputfile is not specified, output will be directed to the terminal display.

(5) Create a new SPLCHK dictionary and add words to it:

```
$ splchk -n [inputfile]
```

Inputfile should contain a list of words, one per line, which are to be added to the new dictionary. If inputfile is not specified, input will be expected from the terminal keyboard.

(6) Replace the incorrectly spelled words in a text file:

```
$ splchk -r inputfile [outputfile]
```

Inputfile will be searched for incorrectly spelled words. If inputfile is not specified, an error message will be displayed, and SPLCHK will end processing.
Outputfile, if specified, will contain the updated, corrected copy of the contents on inputfile. If outputfile is not specified, a new, higher version of inputfile will be created, and output will be directed to that file.

(7) Check the spellings of words, one at a time:

```
$ splchk -w
```

Input is expected from the terminal keyboard.