SPELLING CORRECTION IN USER INTERFACES

IVOR DURHAM, DAVID A. LAMB, and JAMES B. SAXE *Carnegie-Mellon University*

Ivor Durham is currently working in the areas of software reliability, human engineering, and personal computing environments. He is writing a thesis on a methodology for developing fault-tolerant software. David A. Lamb is involved in software engineering, compilers, programming languages, and electronic mail. James B. Saxe is interested in computational complexity and is working on a thesis dealing with transformations of algorithms and circuits.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and in part by the Office of Naval Research under Contract N00014-76-C-0370. J.B. Saxe was supported in part by an IBM Graduate Fellowship.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Authors' Present Address: Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213. CMU-CS-A.ARPA

0001-0782/83/1000-0764 75¢

ABSTRACT: The feasibility of providing a spelling corrector as a part of interactive user interfaces is demonstrated. The issues involved in using spelling correction in a user interface are examined, and a simple correction algorithm is described. The results of an experiment in which the corrector is incorporated into a heavily used interactive program are described. More than one quarter of the errors made by users during the experiment were corrected using the simple mechanisms presented here. From this we have concluded that there are considerable benefits and few obstacles to providing a spelling corrector in almost any interactive user interface.

1. INTRODUCTION

The automatic detection and correction of spelling errors in prose has received a considerable amount of attention (an annotated bibliography is given by Peterson [7]). However, users spend a considerable amount of time typing commands to the user interfaces of programs, and they make typographical errors similar to those made while entering prose. Although designing and implementing a well-engineered and tolerant user interface requires considerable effort, it is possible that some basic techniques, such as correcting the spelling of keywords, can be applied at low cost. In discussing desirable attributes of good user interfaces, Hayes et al. [4] assert that spelling correction in typical interactive programs is straightforward, since usually an error is made in a context where only a limited of keywords (fewer than 100) are applicable. If spelling correction is really "straightforward" then there is no reason why almost all user interfaces should not provide such a facility, even to the extent of retrofitting a corrector into existing programs. Morgan [6] describes a spelling corrector that was implemented in both an operating system and a compiler, for which the statement of purpose was:

> The goal of the proposed spelling correction techniques is, roughly stated, to achieve a proficiency comparable to that of a quick scan of the source program by an experienced programmer who has no knowledge of the program, and who makes no attempt to understand its purpose.

There seem to be very few systems that take advantage of such a facility. The most well known such facility is DWIM (Do What I Mean) in Interlisp [9], whose statement of purpose is almost the same as Morgan's.

The purpose of our investigation was to learn what issues distinguish spelling correction in user interface applications from the more general problem of manuscript spelling correction and to learn how much of a contribution a spelling corrector could make in a user interface. We also wanted to get a realistic picture of the errors users really make and to supplement the data presented by Damerau in 1964 [1]. To this end we considered what characteristics of a spelling corrector for an interactive program were desirable and developed a variation on Damerau's algorithm meeting these requirements. We incorporated the spelling corrector into the command table module used by the RdMail electronic mail system [5], which is in heavy daily use by a community of several hundred researchers. RdMail has a conventional typescript-oriented command language, where most commands consist of a verb followed by a sequence of arguments. Most user terminals are low-bandwidth "glass teletypes," which can display 24 lines of 80 characters each. To learn about the issues and to find out what kinds of mistakes are made by users, we collected data on the accuracy of the spelling of the keywords entered by users and on the ability of the program

to offer corrections when keywords were not recognized immediately. We collected general information about the use of the command interface and recorded specific information about the keywords that were not recognized and what corrections were made.

Throughout the paper we illustrate issues with examples of a user interacting with RdMail. Output appears in boldface type to distinguish it from user type-in. The symbol ^{CR} represents the carriage return key. Italics are our comments and explanations, not part of the typescripts.

2. DESIGN ISSUES

We were prompted to look for a new correction algorithm because of several differences between correcting spelling in general manuscripts and correcting spelling in user interface applications. Design considerations for the spelling corrector fell into two categories: those affecting the design of the algorithm, and those affecting its use in an interactive system.

2.1 Algorithm Design Issues

We chose the same set of assumptions about typographical errors as Gorin did for the PDP-10 SPELL program ([3]; see also [7]). We assume that there is exactly one error in the symbol to be corrected and that the error arises from one of the four causes that account for over 80 percent of spelling errors [1]:

(1) transposition of two adjacent letters
(2) one letter wrong
(3) one extra letter
(4) one letter missing.

These errors are illustrated in Figure 1.

To minimize the difficulty of modifying programs to use the new corrector, we decided to transparently replace a library keyword lookup routine with a version that did spelling correction. The original symbol table module accepts as parameters an unsorted vector of strings and a single probe string to match against the elements of the vector. The lookup algorithm allows the probe to be a unique initial substring of a table entry and reports an error when the probe is ambiguous.

The specification of the original library module illustrates three differences between spelling correction in manuscripts and spelling correction in user interfaces:

(1) It is common to allow abbreviations in a user interface to minimize typing.
(2) The probe being looked up in a table may match several entries.
(3) Affix (suffix and prefix) analysis is not necessary since the legal symbols come from a very limited vocabulary.

2.2 User Interaction Issues

A number of issues arise when a user interface is supplemented with a spelling corrector. These issues concern the interactions with the user when spelling correction is attempted and can interfere with the user's ability to work with the interface.

If the user's symbol contains only one character, the "extra-letter" test would omit this character from the dictionary search and would therefore match all the words in the dictionary (or none of them, depending on the semantics of an empty string), which does not help the user. Similarly, the "wrong-letter" test would match any word beginning with any other character in the symbol alphabet. We chose to report that no match had been found in this case.

If the user's symbol contains only two characters, the various tests for diagnosing errors may still produce a substantial number of possible matches. The "transposition" test and "missing-letter" test are reasonable and behave the same for both two-character symbols and longer symbols. However, the extra-letter and wrong-letter tests can produce a large number of possible matches. Suppose the user's symbol is *xy*; then the extra-letter test would match all symbols beginning with either *x* or with *y*. The wrong-letter test would match all words beginning with *x*? and ?*y*, where ? matches any character. The designer must decide whether the size of the set of possible matches is sufficiently small to permit the user to choose one of them, or whether to behave as though no match had been found. We initially chose to omit both the extra-letter and wrong-letter tests for two-character symbols. After a few months of operation we included the wrong-letter test and received many complaints that the spelling corrector offered too many choices, most of which were quite unexpected; this supports the original decision to omit the test.

Suppose that the correction algorithm finds exactly one matching symbol. Is it safe to assume that the correction is accurate? In general, the answer is no, because the user may have made an error (or multiple errors) not detected by the four tests. An example of this is to omit the space between two keywords. The designer's decision must be based on the consequences of using the symbol in error. The following example is quite harmless:

<-hlep^{CR}
% I assume you mean 'Help' instead of 'hlep'.
Help text is output here.

However, the consequences of assuming the accuracy of a particular correction may be much more serious, as we demonstrate in the following contrived example:

<-overwite^{CR}
% I assume you mean 'Overwrite' instead of 'overwite'.
Program proceeds to expunge deleted messages.

The unfortunate user did not mean "Overwrite"; he actually meant

<-dover wite^{CR}
Program sends file "WITE" to the Dover xerographic printer.

In RdMail this problem is avoided by requiring confirmation before some irreversible action is taken, even if the user did not make a spelling error; these mechanisms are entirely outside the spelling corrector.

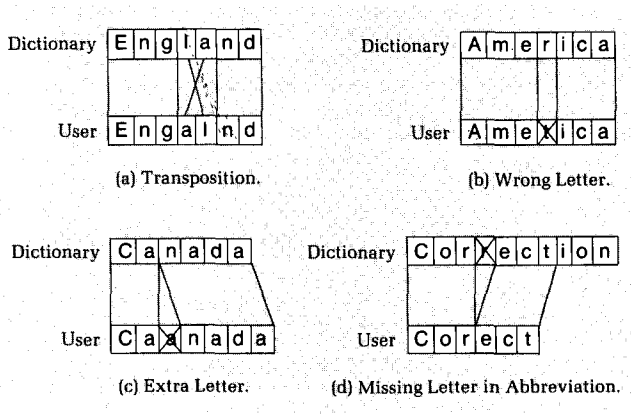

FIGURE 1. Keyword Matching Patterns.

Suppose next that the engineering decision is to require confirmation of all spelling corrections. In a human-engineered system, the actions taken by the system should require little effort by the user. For example, in offering particular default answers to questions, the common response to accept the default is simply to hit one key, carriage return:

<-List All[CR]
onto file? [LPT: MAIL]:[CR]
Program lists all messages onto the line printer.

In the previous example we see that the user can still suffer from prior training to hit [CR] in response to defaults offered by the system:

<-overwite[CR]
% Do you mean 'Overwrite' instead of 'overwite'? [Yes]:[CR]
Program proceeds to expunge deleted messages.

Clearly the default response when the user typed hlep instead of Help could safely have been Yes, while in the Overwrite example it would have been dangerous. It is important that the default responses for spelling correction be consistent to prevent serious mistakes. Although it might be more frustrating for the user to have to say y[CR] to accept Help for hlep, it is certainly better than losing information in the Overwrite example. Hence, the safe version of the previous example is

<-overwite[CR]
% Do you mean 'Overwrite' instead of 'overwite'? [No]:[CR]
No damage this time.
<-dover wite[CR]
System prints file WITE.

A reasonable compromise may be to assume the accuracy of corrections for which the consequences are reversible (flagging such dictionary entries, for example) and to request confirmation of those for which the consequences are not reversible. A simple variant of this last option is used in the current version of RdMail. Confirmation is required for corrections made in some dictionaries but not others. For example, main command corrections are assumed to be accurate since all actions are further confirmed or are reversible. However, confirmation is required when an error is made in naming a program for RdMail to run as a subjob. To assume the wrong program name could have irreversible consequences, such as deleting files. A RdMail user may set an option to always request confirmation of spelling corrections.

When the correction algorithm finds more than one matching symbol in the symbol table, the designer must decide whether or not the user should be given the opportunity to select the correct symbol from the smaller set of matching symbols. He must also decide whether to invest the effort in further reducing the size of the set of matching symbols by using heuristic factors, such as the relative positions of characters on the keyboard. We included no such heuristics and simply offered the user all matching symbols. The data collected during our experiment showed that between 2 and 10 alternative corrections were offered, with the majority of cases producing between 2 and 4 alternatives.

<-ecx[CR]
% 'ecx' could be any one of the following:
Echo, Exit
Which one do you mean? [None of the above]:[CR]

An alternative strategy for handling common ambiguities is to provide preferred disambiguations. For example, in RdMail "A" is presumed to mean "Answer" instead of "Accept," "Alias," or "Allocate." This is handled by a mechanism outside the spelling corrector: the single-character command A is added to the command table as a synonym for Answer.

Finally, what should be done if there is a problem with the symbol supplied by the user in response to the question, "Which one do you mean?" Some of our users suggested that spelling correction should be applied recursively, but others wanted to be able to type in the name of the command they had meant initially. For example, when a user typed de instead of ed for Edit, the program offered as alternatives only those commands for which de is an ambiguous abbreviation; the user would probably prefer to respond with Edit even though the symbol is not in the set offered by the program. Clearly a combination of the two could be applied. For example, if the program were to apply spelling correction first and find that that still didn't produce an unambiguous symbol, then it could look for the new symbol in the original symbol table rather than in the set of possible corrections. On the other hand, this would require the user to maintain a complex model of what the corrector is doing. For our experiment, we chose the simple expedient of forcing the user to get the symbol right rather than making any attempt to correct the correction. The program simply repeats the question:

<-de 201[CR]
% 'de' could be any one of the following:
DeAllocate, Debug, DeClassify, Delete
Which one do you mean? [None of the above]: dle[CR]
% 'dle' is not an option.
Which one do you mean? [None of the above]: del[CR]
Program deletes message 201.

The action taken at a user interface when all attempts fail to produce a unique symbol is not specific to those interfaces that use spelling correction. However, a few simple actions should be mentioned. First, the command containing the erroneous symbol may be aborted. Having tried our best to make sense of the command, we must ultimately give up since it will be much easier for the user to express himself more accurately. This approach was used in RdMail.

<-Aaaarrgghh![CR]
? No such command as Aaaarrgghh. Type ? for help.

A slightly more sophisticated approach is to ask the user to correct the symbol and then attempt to continue with the command (notice that two errors are detected in the command):

<-haeders from Robertson intersect week "May 16"[CR]
% I assume you mean 'Headers' instead of 'haeders'.
? 'week' is not a Message Sequence keyword.
Message Sequence keyword [Abort command]: since[CR]
Program lists headers of messages from Robertson since May 16.

This mechanism was added to RdMail after our experiment.

3. THE CORRECTION ALGORITHM

The comparison of two symbols (the user's and a dictionary symbol) is done in three parts, as illustrated by the three divisions of each example in Figure 1.

(1) Find the common initial substring (i.e., up to the first difference). Case distinctions in letters may be ignored; the algorithm must find all possible matches for the symbol not found by the initial search.

(2) Examine the next pair of letters for a transposition error.

(3) Match the tail substrings. If the previous step suggested that two characters had been transposed, omit two characters from both symbols and match the remaining substrings. For the wrong-letter test omit one character from each symbol and match the tail substrings. For the extra-letter (missing-letter) test, omit one letter from the user's (dictionary) symbol and match the remaining tail substrings.

The matching steps are repeated for each symbol in the dictionary. Each dictionary symbol that is matched is added to a set of possible corrections for the user's symbol. If there is only one member in the set when all of the symbols in the dictionary have been examined, that symbol may be offered as the correction. If there are several symbols in the set, the user may be asked to select one.

We originally tried the SPELL program's strategy of searching the symbol table for each string that could be transformed into the user's symbol by one of the four kinds of errors. This was far too slow with the original library lookup algorithm.

Using only the lengths of the user's symbol and the dictionary symbol, two optimizations can be made to avoid unnecessary string comparisons:

(1) If the length of the user's symbol exceeds the length of the dictionary symbol by more than one character, no match is possible with the above algorithm, so the "no match" result can be returned immediately. (This assumes that the string length is readily available.)

(2) If the first difference found is in the last character of the user's symbol, the extra-letter test would discard the character and therefore convert the user's symbol into a matching initial substring of the dictionary symbol.

This algorithm requires no intermediate string construction. The only additional storage required is for the set of matched dictionary entries, which can be represented as a vector of Booleans with one element per dictionary element. At the end of the algorithm the set of matching symbols is identified by all *true* elements in the set vector. The algorithm also has the advantage of being trivial to implement. The most expensive components are the low-level functions that locate the first difference and match substrings. In our experimental implementation both of these functions were written in assembly code using straightforward character-by-character comparisons.

An example implementation of this algorithm in Ada[1] is given in the Appendix. This implementation exploits Ada's facility for dynamically dimensioned arrays. In languages which lack this facility, other data structures, such as linked lists or large fixed-size arrays, can be used instead.

3.1 Implementation of the Corrector

The main spelling correction algorithm was implemented in SAIL[8], an Algol-based language that provides strings as a primitive data type. There is an extensive library of SAIL functions at Carnegie-Mellon University that includes a command table abstraction and that uses a more primitive symbol table abstraction. The command table module was changed to call the spelling corrector when a keyword was not found anywhere in the symbol table.

To handle multiple matching symbols, the corrector builds a table (vector) of string pointers and calls a subroutine which asks the user to select the correct keyword. The subroutine forces the user to be accurate in selecting one keyword from the set offered. No attempt is made to correct the spelling of the keyword selected.

3.2 Performance

The following informal analysis shows that the spelling correction algorithm is quite adequate for our requirements even though it clearly is not optimal for the general correction application in prose. What it lacks in performance is returned in simplicity that facilitates its introduction in a wide variety of applications.

To correct one symbol given a dictionary of N symbols, our algorithm performs N initial substring matches and at most $4N$ tail substring matches. (The transposition tail match is performed only if the "transposition" test succeeds.) At worst, this is equivalent to $4N$ equality string matches, plus a small constant overhead per dictionary element.

To get a more concrete measure of the cost of spelling correction, we performed some measurements of the algorithm's running time. Measurements were taken on a lightly loaded DECsystem-10 KL-10 processor[2] (the same one used in the RdMail experiment described in Section 4). In each case the data were obtained by running 5000 tests in a loop, subtracting the original value of the system clock from the final value, subtracting loop overhead, and dividing by 5000. Lookups were done on a table of 66 entries, a subset of the main command table from the RdMail program. The original command table had 77 entries; we eliminated 5 punctuation character commands and 6 commands where transposing the first two letters results in an ambiguity. Measurements were taken of

(1) the time taken by the original library module (without spelling correction) and by the new module (with spelling correction) to look up a correct entry.

(2) the time taken by each of the two modules to decide a probe is not in the table. Five sources of failing keys were chosen:

   (a) character strings of the form *aaaaa*, *bbbbb*, and so on
   (b) failing keywords collected during the experiment described in Section 4; all keys longer than two characters were included
   (c) the numeric keys from the experiment
   (d) the alphabetic keys from the experiment
   (e) strings of the form KEYxxx, where KEY is a command from the main table

   These times do not include the time taken to print an error message; printing times are reported separately. We also report the difference between the new and the old lookup times, which represents the time taken by the spelling corrector, and this difference divided by the table size, which roughly represents the cost per table entry. The last number varies because of the optimizations mentioned in Section 3, which can reject some symbols quickly.

(3) the time taken to handle a transposition of the first two characters of a command: a command was chosen and its first two letters were transposed before calling the routine, which resulted in a lookup failure for the original module; the new module corrects this error

(4) the time taken to print a message of the form "I assume

[1] Ada [2] is a registered trademark of the U.S. Department of Defense (OUSDRL-AJPO).

[2] DECsystem-10 is a trademark of Digital Equipment Corporation.

TABLE I. Measurements of the Corrector

Test	Time (ms)			
	Original lookup	New lookup	Difference	Per table entry
Succeeding	1.781	1.797		
Failing				
aaaaa, etc.	3.100	24.728	21.628	0.328
real data	3.123	22.060	18.937	0.287
real numeric	3.162	25.613	22.451	0.340
real alphabetic	3.122	21.943	18.821	0.285
KEY *xxx*	2.990	12.385	9.395	0.142
Transposition				
failing	3.221	—		
correcting error	—	28.384		
Print "I assume . . ." message	5.749			
Print ". . . not a command" message	4.594			

you mean *X* instead of *Y*"

(5) the time taken to print a message of the form "*X* is not a command."

The results are illustrated in Table I.

We conclude that the algorithm described above is a good choice for those applications, such as user interfaces, in which the size of the dictionary is quite small and abbreviations must be handled. For the library symbol table module used in the experiment, we cannot do much better since the specification for the lookup routine does not require that the table be sorted.

4. THE EXPERIMENT

The principal purpose of the informal experiment described in this section was to learn how useful a spelling corrector might be in an interactive user interface. In asking about the errors made by users, our attention is confined to finding symbols in the various symbol tables used by the program; we explicitly exclude semantic and syntactic errors in composing commands except as they are detected by failing to find a symbol in a particular table. The efficacy of the correction facility depends on the variety of errors users make and their respective frequencies. The rate of errors made determines the overall cost of the spelling correction facility. We need to determine what proportion of those errors can be attributed to typographical errors that may be recovered by the corrector. Learning what other errors are made may suggest other ways to improve the tolerance of the user interface to user errors.

The RdMail Message Management System [5] was used for the experiment. We describe the program and the information we recorded from it in the next section. Then we present the results of the experiment and offer a brief evaluation of the effectiveness of our mechanisms in the light of those results.

4.1 The RdMail System

RdMail is an interactive electronic-message management system that provides facilities for the composition and transmission of messages among users of a network of computers. Messages received can be classified, answered, and filed conveniently. RdMail commands are sequences of keywords and parameters, where the parameters are usually numbers or quoted strings. Users may operate on sets of messages by specifying the particular messages by number, attribute (such as date of arrival, name of originator, or subject), or user-defined classification. For example,

```
<-headers meetngs intersect (since "Jan 1") intersect 50:175^CR
% Do you mean 'Meetings' instead of 'meetngs'? [No]: y^CR
```

produces a brief identification of all messages between 50 and 175 that arrived since January 1 and were classified as "meetings." Since the user defines the names of classifications, such as "meetings," the set of symbols in the table used in the parsing of message set specifications is dynamic.

The user leaves RdMail either temporarily to use an editor to alter the composition of a message or permanently to return to the system's executive program. For the purposes of the experiment, a *session* encompasses only those commands that are given before the user leaves RdMail for any reason. Hence one user session in which a message arrives, an answer is composed and edited before being mailed, and new mail is sent to other people would be considered two sessions in the collected data: the first before entering the editor, the second after returning from the editor and before leaving the program permanently.

For the duration of the experiment, RdMail forced users to confirm *all* corrections suggested by the program. This was the only alteration made in the RdMail user interface.

4.2 The Data Collected

Because of the sensitive position of RdMail in communicating between users we recorded no data that could be traced directly to particular users. This anonymity was also important in reducing the probability of users becoming self-conscious about making typographical errors and taking more care than usual. We felt morally obliged to warn people that we were performing an experiment and to give them the option of running a different program to avoid participating in the experiment. Warning users that the experiment was to be conducted encouraged a few to entertain us with some colorful, if illegal, keywords. In some cases it was apparent that users were probing the limits of RdMail's correction facility. However, we cannot be sure of any particular user's intention and have therefore included the apparently intentional errors in our results.

The data recorded for each RdMail session were

(1) the number of commands given to RdMail (both from the keyboard and from preexisting files)
(2) the number of keywords for which RdMail searched symbol tables and the number of those keywords that were not found or were ambiguous.

For each symbol that was not uniquely matched in a particular symbol table, a detailed record was made including

(1) the symbol the user provided
(2) the correct symbol, if any, as confirmed by the user
(3) the number of possible corrections for the symbol identified by the spelling corrector
(4) the identity of the symbol table
(5) the approximate execution time taken to identify the corrections that could be made (rounded to the nearest millisecond).

Gathering additional data, such as the entire command line containing each unrecognized symbol, would have aided us in determining the causes of uncorrected errors. We decided not to do this because of our respect for privacy.

```
<-headers from Bovik intersect subejct "pay raise"^CR
```

We shouldn't learn that Bovik is up for a pay raise just because someone misspelled "subject."

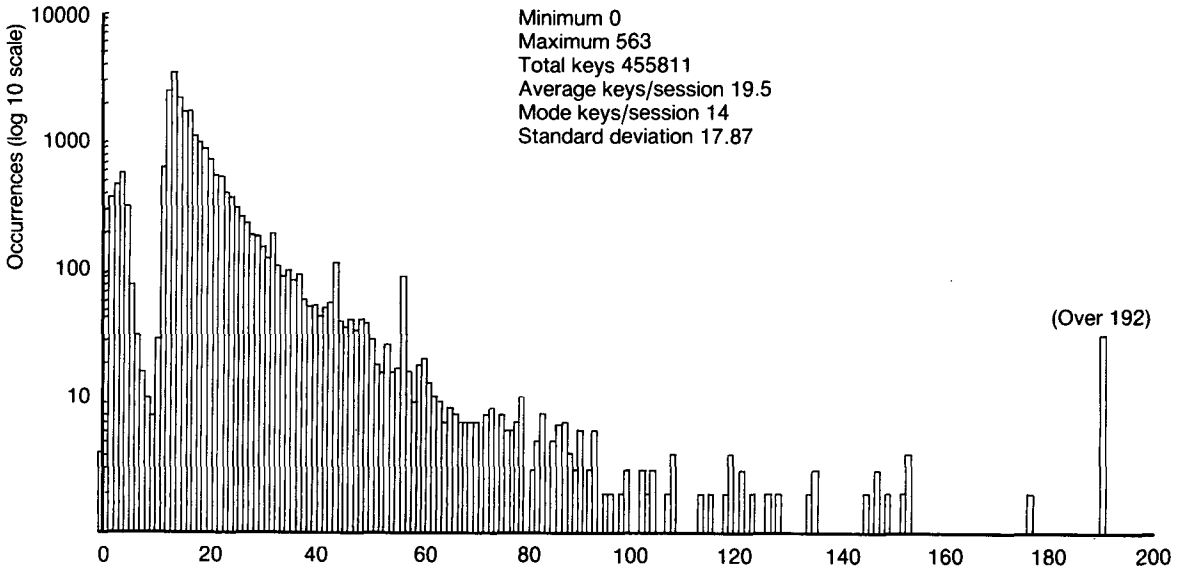

FIGURE 2. Keywords in a Session.

The reason we chose RdMail as our experimental vehicle in spite of this inconvenience is that, with the possible exception of various operating systems, RdMail has by far the most heavily used "command line" style interface in our environment. The other heavily used programs are either compilers, which are not interactive, or text editors, which use mostly single-character commands.

4.3 Usage Statistics

The experiment ran for 41 days during which time a total of 23,361 RdMail sessions were recorded. RdMail processed a total of 145,972 commands during the experiment; 140,038 from terminals and 5934 from command files. Data from batch jobs were discarded because we were interested only in human typographical errors, not in general RdMail use. RdMail handled a total of 455,811 keywords during the experiment, averaging 3 per command. The distribution of numbers of keywords in sessions is shown in Figure 2.

The running time for identifying the set of possible corrections varied considerably, ranging up to 31 ms with an average of 9.5 ms, but with a relatively large standard deviation, $\sigma = 9.5$ ms.[3] The total time used by the spelling corrector over the 41 days of the experiment was 19.2 seconds, an average of 468 ms per day.

4.4 Results

During the 41 days of our experiment, RdMail encountered 2527 erroneous, i.e., not uniquely identifiable, keys (0.554 percent of all symbols entered). Due to an oversight in the data-collection routines, we cannot determine how many keys came from command files. Four percent of commands came from command files. Even if the number of keywords per file command were an order of magnitude greater than the average number of keywords for all commands, the error rate for manually entered keys would be only 0.934 percent, which is still very small. Actual error rates may be higher, since we cannot tell how often a user noticed an error and corrected it manually (by backspacing over the error or deleting the input line, and retyping) before hitting carriage return to enter the command. The erroneous keywords were recorded in two different classes: keywords not found in the symbol table and keywords that were ambiguous. The distribution of these errors is shown in Table II.

By examining the data collected for each erroneous key, we arrived at the taxonomy of errors shown in Figure 3. The percentage figure in parentheses after each class of error gives the size of that class in relation to the entire class of 2527 recorded instances of erroneous keys.

Corrected Errors (27 percent). *Transposition error corrected, Missing letter restored, Wrong letter corrected, and Extra letter removed* (16 *percent*); *Ambiguity resolved* (11 *percent*). The error recovery mechanism offered potential corrections (24 percent) or disambiguations (20 percent) for 44 percent of all erroneous keys. However, users did not always accept corrections and disambiguations when they were offered. Only 56

TABLE II. Keywords in Error

Unmatched keys per session	Number of sessions	Ambiguous keys per session	Number of sessions
0	21800	0	22905
1	1293	1	423
2	181	2	28
3	44	3	3
4	27	4	2
5	7		
6	2		
7	3		
8	0		
9	2		
10	1		
40	1		
Total unmatched keys	2031		
Total ambiguous keys	496		

[3] The average time spent in the spelling corrector for the subset of the data used as "real keys" in Section 3.2 was 19.7 ms. This agrees well with the spelling corrector cost (19.1 ms) shown in Figure 2. Many erroneous keys were processed more quickly because they were shorter than three characters or because they were looked up in small tables.

All erroneous keys (100%)
- Corrected/disambiguated keys (27%)
  - Ambiguity resolved (11%)
  - Typographical (16%)
    - Transposition error corrected (2.7%)
    - Missing letter restored (4.8%)
    - Wrong letter corrected (4.5%)
    - Extra letter removed (4.0%)
- Uncorrected keys (73%)
  - Alphabetic (46.6%)
    - One character (10.4%)
    - Two characters (9.9%)
    - Three or more characters (26.3%)
      - Typographical? (2.9%)
        - Missing space?
        - Missing carriage return?
        - Missing slash?
        - Control key?
        - Typeahead?
        - Multiple typo?
        - Miscellaneous typo?
      - Nontypographical? (23.4%)
        - Intentional error? (3.2%)
        - Good correction or disambiguation rejected? (0.4%)
        - Syntax or vocabulary error? (19.8%)
  - Nonalphabetic (26.4%)
    - Control character (12.9%)
    - Number (4.6%)
    - Punctuation (8.9%)

FIGURE 3. Taxonomy of Errors. Speculative classifications are marked with "?".

percent of the ambiguities detected were resolved by the user accepting one of the alternatives offered, and users accepted spelling corrections in only 66 percent of the cases where one or more potential corrections were offered. The errors thus resolved accounted for 27 percent of all erroneous keys. For 13 percent of all erroneous keys, the corrector offered a single correction that was accepted by the user.

In some cases, accurate disambiguations or corrections may have been rejected accidentally. We are unable to say how often this happened because the privacy constraints on our experiment prevented us from recording sufficient information to determine which rejected disambiguations and corrections were in fact accurate.

Errors with two or more explanations were assigned to the first of the above categories into which they fit. For example, if the erroneous key dle was corrected to Deleted, we accounted for the error as the transposition of "l" and "e," rather than as the omission of an "e" or the inclusion of a spurious "l."

Uncorrected Alphabetic Keys (73 percent). Of the 1845 uncorrected erroneous keys, 1179 (46.7 percent of all erroneous keys) were "alphabetic," that is, consisted of a letter followed by zero or more letters or digits. Of these, 264 (10.4 percent of all erroneous keys) were single letters, 249 (9.9 percent) were only two characters long, and 666 (26.4 percent) were 3 or more characters long. We manually classified the 666 "multicharacter" (>2 character) uncorrected alphabetic keys. Since we had to rely on educated guesswork for this classification, it is possible that we incorrectly classified some of the keys. We have indicated this possibility by placing question marks by the names of the manually generated subclasses.

Typographical Errors (2.9 percent). *Missing space, Missing carriage return, Missing slash, Control key, Typeahead, Multiple typo, Miscellaneous typo.* We attributed 74 of the 666 erroneous multicharacter alphabetic keys (2.9 percent of all erroneous keys) to typographical errors of sorts not corrected by our algorithm. Perhaps the most obvious sort of error in this category is the omission of a space between two keywords (e.g., typing numnew instead of num new to ask for the message numbers of all new messages, or hdel instead of h del to ask for the headers of all deleted messages). A similar kind of error, but one whose existence we might not have guessed without seeing some examples, is the missing carriage return. An example is the key exitbb, almost surely typed by a user who intended to type an Exit command to leave RdMail and then type BB to the operating system to read an electronic bulletin board. In one case we diagnosed an erroneous key as resulting from a missing "/." On our system, the control character CTRL-S is used to suspend output to the terminal, an action that is useful to prevent long messages from scrolling off the screen faster than the user can read; typing a CTRL-Q causes output to resume. If the CTRL key on a terminal is broken, or if the user doesn't have his finger on it—some of our terminals have keyboards with REPEAT in the same position where others have CTRL—the result may be an erroneous key such as ssssty (instead of ty to type a message). Another feature of our system is that terminals run in full duplex mode, allowing the user to enter additional commands while waiting for the machine to respond to earlier commands. Since such "typeahead" may not be echoed immediately or may be echoed in the middle of a lot of output, it is possible for a user to forget how far ahead he has typed. An example of an erroneous key that is probably due to this phenomenon is typetype. Presumably the user keyed in the command Type while waiting for the previous command to finish, then forgot that he had done so and keyed it in again. We attributed eight erroneous keys to multiple typos. Two examples are aner (instead of Answer to reply to a message) and hbok (instead of h book to type the headers of all messages in the user-defined message class book). Finally, there were several erroneous keys which appeared to result from problems with the mechanics of keying in commands but for which we could not confidently specify a most probable cause. It is interesting to note that 353 of the corrected keys (excluding disambiguated keys) were three or more characters long. Assuming that our count of 74 typographical errors among the uncorrected multicharacter alphabetic keys is accurate, this means that 83 percent (353 out of 353 + 74 = 427) of all typographical errors resulting in multicharacter alphabetic erroneous keys were in the four classes handled by the corrector. This is an agreement with Damerau's [1] experience that these four classes account for over 80 percent of all spelling errors.

Nontypographical Multicharacter Alphabetic Erroneous Keys (23.4 percent). *Intentional error, Good correction or disambiguation rejected, Syntax or vocabulary error.* In addition to the 74 multicharacter alphabetic keys that we could diagnose as typographical errors, there were 80 keys (3.2 percent of all erroneous keys) that appeared to be intentional errors and 10 cases (0.4 percent) in which we were reasonably confident that accurate corrections or disambiguations were rejected by users. The intentional errors included messages to the authors of the spelling corrector (e.g., hithere and

doyoureallymeanyoucantfigureouthbok), strings which appeared to result from use of the keyboard as a pacifier (e.g., kkklknlkn), and a sequence of 28 consecutive misspellings of the command Put (up, tup, tpu, sput, etc.)—presumably generated by a user who was probing (i.e., playing with) the spelling corrector. This leaves 502 (19.9 percent) legitimate multicharacter alphabetic erroneous keys, which we must presume were due to errors above the typographical level—i.e., syntax and vocabulary errors. Broadly speaking in these cases the user either forgot the appropriate keyword, used a keyword that would have been recognized in some other context, or induced a parsing error by omitting a symbol, thereby leaving an operand keyword where an operator was expected (or vice versa). The following examples are typical:

<-headers from Durham since 3-mar[CR]
? Illegal message sequence at "SINCE" – junk at end
FROM DURHAM SINCE 3-MAR
↑

<-headers from Durham intersect since 3-mar[CR]

Program prints headers of messages from Durham dated later than March 3.

<-kjob/a[CR]
? No such command as kjob. Type ? for help.
<-exit[CR]

EXIT

.kjob/a[CR]

Logged off CMUA.

Nonalphabetic Erroneous Keys (26.4 percent). *Control character, Number, Punctuation.* The nonalphabetic erroneous keys included 325 control characters (12.9 percent of all erroneous keys), 116 numbers (4.6 percent), and 225 punctuation marks (8.9 percent). Among the control characters, the most common by far was CTRL-S, which occurred 181 times (7.2 percent of all erroneous keys). As we mentioned earlier, this character is used on our system to suspend output to a terminal temporarily. Normally, the user types CTRL-Q to cause output to resume. However, typing a second CTRL-S while output is suspended will cause output to resume, but the operating system will pass the second CTRL-S to the program's input stream. If a user types CTRL-S, but output doesn't stop immediately (because the load on the system is impairing response time), he or she may type a second CTRL-S, thereby inadvertently sending a CTRL-S to RdMail. We believe that this phenomenon accounts for all, or almost all, the observed occurrences of CTRL-S as an erroneous key. In some cases CTRL-S might have been intended as SHIFT-S, but these cases alone can hardly account for the great frequency of CTRL-S compared with other control characters. Given RdMail's command syntax, we would have expected numbers and punctuation marks to appear most frequently as erroneous keys in the middle of long commands. Surprisingly, 84 percent of the numbers and 51 percent of the punctuation marks, as well as 92 percent of the control characters other than CTRL-S, occurred as the first symbols of the commands in which they were detected as erroneous keys. We have no solid explanation for this phenomenon.

4.5 Evaluation

Our mechanism handled 27 percent of the erroneous keys entered during the experiment. Examination of the remaining 73 percent led us to wonder what other mechanisms might permit further corrections while retaining the typescript-style interface. Most of the other errors seemed specific to the operating system (TOPS-10) or application (RdMail). Although there does not seem to be a mechanism as general as the spelling corrector for handling these errors, we believe that developing an "expert" level of friendliness requires paying attention to this sort of detail.

Since the ambiguous key "D" was almost always disambiguated into "Delete," adding D to the main command table as a synonym for Delete would remove 3.7 percent of the errors. Ignoring the character CTRL-S, or treating it as a space, could eliminate 7.2 percent of the errors. Since we believe most of these occur because of attempts to suspend typeout, this seems reasonable. Ignoring all control characters could account for a further 5.7 percent, but further study is needed to determine why these errors occur.

Errors caused by typeahead might be reduced by not echoing characters until the application requests input, as is done on TOPS-20. This might actually increase error rates, since users would not be able to see their typeahead. Our data indicate that typeahead errors are very infrequent.

A portion of the syntax and vocabulary errors (19.8 percent of all erroneous keys) and numeric errors (4.6 percent) may be due to omitted keywords, or to the user forgetting the context. These errors may be amenable to the techniques described by Hayes et al. [4]. Some syntactic errors might be handled by the recovery techniques used in compilers, or might be eliminated by modifications to the grammar. For example, after the experiment we made a small modification to the grammar for RdMail message sequences so that a user may omit the keyword "intersect."

Finally, there are some errors that do not seem to admit any reasonable automatic recovery. For example, if a user tries to classify a message as "ICs" (a user-defined class for messages regarding integrated circuits), when the name of the class is actually "chips," the best that can be done is to allow the user to choose among the names of all the classifications.

5. CONCLUSION

The spelling corrector offered a unique acceptable correction for 13 percent of the keyword errors detected during the experiment. In a further 3 percent of the cases it found multiple possible corrections, one of which was accepted by the user. Allowing the user to correct ambiguities manually fixed a further 11 percent of the keyword errors.

The correction algorithm is very simple to implement and costs us about half a second per day for a heavily used interactive system. The corrector was invoked about 50 times a day at an average cost of about 10 ms. RdMail has since been modified to use the spelling corrector when the user's keyword is an ambiguous abbreviation, as well as when the keyword is not in the symbol table at all. If we project with our data, the invocation rate increases to about 60 times per day. The data clearly support the premise that spelling correction is "straightforward" in user interface applications. The most complex part of the engineering is selecting the behavior of the system with the results of the correction algorithm.

It is interesting that, in response to repeated requests by certain users, the RdMail maintainers have provided options for suppressing all of the extra warnings and confirmations normally produced when some irreversible action is about to occur. Such users are vulnerable when the corrector changes a typographical error into a valid, irreversible command. The mistakes made by experts appear intuitively to be caused by rapid typing and extensive use of abbreviations, while less

experienced users tend to use full command names and make the more common typographical errors.

We have installed the command module that uses the spelling corrector in the standard SAIL library at Carnegie-Mellon University. As a consequence, any program that uses the library module acquires the spelling correction facility the next time that it is link-edited. The number of programs that now routinely provide spelling correction without any action at all on the part of their author or maintainer is growing slowly.

We conjecture that the spelling correction facility and algorithm described in this paper would be equally beneficial in both operating system environments (interactive and batch) and compiler applications, where computing resources might be conserved by continuing computations that might otherwise be aborted, only to be repeated later. In particular, we are somewhat surprised that the work described by Morgan in 1970[6] has not found wider application today. We foresee no significant technical difficulties to implementing our algorithm in a variety of languages. Perhaps our results can convince programmers to provide this simple, cheap, and effective facility in new and even existing user interfaces.

Acknowledgments. This work began in response to a suggestion by Hayes et al. that spelling correction of keywords in programs like RdMail ought to be easy [4].

RDMail was originally written by Philip Karlton at Carnegie-Mellon University. It was nursed through adolescence to maturity by a series of dedicated people including Mark Sapsford, Craig Everhart, Philip Lehman, and David Lamb. We are indebted to our user community at Carnegie-Mellon for allowing us to conduct the experiment and for providing immediate and high quality feedback on the improvements made to RdMail. Craig Everhart shared with us his considerable expertise to overcome some intricacies of our operating system and gave us valuable advice on the design of the experiment. Mark Sherman helped us to persuade the Intermetrics Ada system that our example implementation of the corrector was indeed good, legal, and operational Ada code. Comments from Jon Bentley, Bob Chansler, Craig Everhart, Phil Hayes, Anita Jones, Anne Rogers, and Mary Shaw helped us to improve the clarity of this paper. Finally, we are indebted to Gorin's SPELL program which did a fine job of correcting the typographical errors in our manuscript.

APPENDIX.
Example Implementation of the Spelling Corrector

The following Ada implementation illustrates the functions required for the spelling correction algorithm. The code was compiled by the Intermetrics Ada Prototype Compiler and executed on a DECsystem-20.[4] This example uses the 1980 version of Ada, since a compiler for 1982 Ada was not available to us at the time of publication.

```
-- Example implementation of the Spelling Corrector in DoD Ada.
-- This code is operational. However, to improve the clarity of this
-- example, we have omitted the detailed interactions with the user.
-- We have excluded the routines that interact with the user (User_
-- Accepts, User_Selects) and have commented out their invocation.

package Spelling_Corrector is

Not_Correctable: exception; --Raised if no corrections found.

type Symbol_Table is array(integer range <>) of string(1..32);

-- The Correct_Spelling function delivers the index in the table of
-- the corrected symbol or raises the Not_Correctable exception.

function Correct_Spelling
  (ST: in Symbol_Table;
   User_Word: in string;
   Assume_Correct: in boolean) return integer;

end Spelling_Corrector;

with text_io;      --Need an I/O package
use text_io;

package body Spelling_Corrector is

-- The Same_Character function returns true iff the two characters
-- are the same when case-differences are ignored.

function Same_Character (A, B: in character)
  return boolean is
    Folded_A, Folded_B: integer; --Case folded character positions
    Case_Difference: constant integer
      := character'POS('a')-character'POS('A');
  begin
    Folded_A := character'POS(A);
    if A in 'A' .. 'Z' then --Upper to Lower case conversion
      Folded_A := Folded_A + Case_Difference;
    end if;
    Folded_B := character'POS(B);
    if B in 'A' .. 'Z' then --Upper to Lower case conversion
      Folded_B := Folded_B + Case_Difference;
    end if;
    return Folded_A = Folded_B;
  end Same_Character;
-- The First_Difference function locates the first character position at
-- which the two parameter strings differ (ignoring case distinctions).
-- Zero is returned if either string is empty.
function First_Difference (A, B: in string) return integer is
    Last_Index: integer;
  begin
    if A'LENGTH <= B'LENGTH then --Find shorter string
      Last_Index := A'LENGTH;
    else
      Last_Index := B'LENGTH;
    end if;
    if Last_Index = 0 then --One string is empty
      return 0;
    end if;
    for i in 1 .. Last_Index loop
      if not Same_Character(A(i), B(i)) then
        return i;
      end if;
    end loop;
    return Last_Index + 1;
  end First_Difference;

-- Function Match_Substring returns true iff the second string (B) is
-- an initial Substring of the first string (A). A is considered to begin at
-- index First_A and B is considered to begin at index First_B.

function Match_Substring
  (A: in string; First_A: in natural;
   B: in string; First_B: in natural) return boolean is
  begin
    if First_B > B'LENGTH then
      return true;      --B is empty sub-string
    elsif (First_A > A'LENGTH) or
        ((B'LAST-First_B) > (A'LAST-First_A)) then
      return false;     --A is empty or B is too long.
    end if;
```

[4] DECsystem-20 is a trademark of Digital Equipment Corporation.

```
    for i in 0 .. (B'LAST-First_B) loop
      if not Same_Character(A(i + First_A), B(i + First_B)) then
        return false;
      end if;
    end loop;

    return true;
  end Match_Substring;

-- Function Possible_Correction returns true iff one of the four tests
-- applied to the user word yields the dictionary word.

function Possible_Correction
  (Dictionary_Word, User_Word: in string) return boolean is
  Index: integer;
  begin
    -- Heuristic: Can't match if symbol is more than one character
    -- longer than dictionary word.

    if (User_Word'LENGTH - 1) >
      Dictionary_Word'LENGTH then
      return false;
    end if;

    -- Step 1: Find the index of the first different characters

    Index := First_Difference(Dictionary_Word, User_Word);

    -- Heuristic: Assume wrong letter if difference at end of word

    if (Index = User_Word'LENGTH) and
      (User_Word'LENGTH > 2) then
      return true;
    end if;

    -- Step 2: Check for transposed characters & tail match

    if ((Dictionary_Word'LAST > Index) and
      (User_Word'LAST > Index)) and then
      (Same_Character(Dictionary_Word(Index),
                      User_Word(Index + 1)) and
       Same_Character(Dictionary_Word(Index + 1),
                      User_Word(Index)) and
       Match_Substring(Dictionary_Word, (Index + 2),
                       User_Word, (Index + 2))) then
      return true;      --Transposition.
    end if;

    -- Step 3: Apply remaining tail Substring matches

    if Match_Substring(Dictionary_Word, (Index + 1),
                       User_Word, Index) then
                         return true;   --Missing letter.
    end if;

    -- Policy: Don't try other tests on 2-character strings.

    if User_Word'LENGTH = 2 then
      return false;
    end if;

    if Match_Substring(Dictionary_Word, Index,
                       User_Word, (Index + 1)) then
      return true;   --Extra letter.
    end if;

    if Match_Substring(Dictionary_Word, (Index + 1),
                       User_Word, (Index + 1)) then
      return true;   --Wrong letter.
    end if;

    return false;
  end Possible_Correction;

function Correct_Spelling
  (ST: in Symbol_Table;
   User_Word: in string;
   Assume_Correct: in boolean) return integer is
  Match_Count, Last_Match: integer := 0;
  Match_Flag: array (ST'range) of boolean;
  Test_Word: string(1 .. User_Word' LENGTH) := User_Word;
  begin
    for i in ST'range loop
      Match_Flag(i) := Possible_Correction(ST(i), Test_Word);
      if Match_Flag(i) then
        Match_Count := Match_Count + 1;
        Last_Match := i;
      end if;
    end loop;

    if Match_Count = 1 then
    if Assume_Correct then
      Put_Line("% I assume you mean "'& ST(Last_Match) &
               "' instead of "'&Test_Word& "'.");
      return Last_Match;
--  elsif User_Accepts(ST(Last_Match), Test_Word) then
--      --Ask Do you mean 'x' instead of 'y'? question.
--      return Last_Match;
    end if;
  elsif Match_Count > 1 then
      Put_Line("% "'& Test_Word &
               "' could be any one of: ");
--    return User_Selects(ST, Test_Word, Match_Flags);
  end if;

    raise Not_Correctable;
  end Correct_Spelling;
end Spelling_Corrector;
```


REFERENCES

1. Damerau, F. J. A technique for computer detection and correction of spelling errors. *Comm. ACM 7*, 3 (March 1964), 171–176. Presents data attributing over 80 percent of spelling errors to a single transposition, extra letter, missing letter, or wrong letter error.
2. United States Department of Defense. Reference Manual for the Ada Programming Language. Government Printing Office, Washington, D.C., Order L008-000-00354-8, 1980. The 1980 version of the official Ada specification.
3. Gorin, R.E. SPELL: Spelling check and correction program. Online documentation, 1971. Documentation online in file DOC:SPELL.DOC at CMU-CS-A.ARPA. Describes operation of PDP-10 SPELL program. A description of the basic algorithm is also given by Peterson [7].
4. Hayes, P., Ball, J.E., and Reddy, D. R. Breaking the man–machine communication barrier. *IEEE Computer 14*, 3 (March 1981). Discusses desirable attributes of user interfaces.
5. Lamb, D.A. RdMail Message Management System: User's Guide and Reference. Carnegie-Mellon University Computer Science Department, Pittsburgh, Pa., 1980.
6. Morgan, H.L. Spelling correction in systems programs. *Comm. ACM 13*, 2 (Feb. 1970), 90–94. Describes spelling correction techniques for compiler and operating system applications.
7. Peterson, J.L. Computer programs for detecting and correcting spelling errors. *Comm. ACM 23*, 12 (Dec. 1980), 676–687.
8. Reiser, J.F. (Ed.) SAIL Manual. Stanford University Computer Science Department, Palo Alto, Calif., 1976.
9. Teitelman, W. InterLisp Reference Manual. Xerox Research Center, Palo Alto, Calif., 1978. Chapter 17 describes the DWIM (Do What I Mean) facility, which corrects simple programmer errors, including misspelled keywords.

CR Categories and Subject Descriptors: D.1.0 **[Programming Techniques]**: General; D.2.2 **[Software Engineering]**: Tools and Techniques—*software libraries, User Interfaces*; H.1.2 **[Information Systems]**: User/Machine Systems—*human factors.*

General Terms: Algorithms, Human Factors

Additional Key Words and Phrases: spelling, spelling correction, typographical errors, user interfaces, interactive programs

Received 7/82, revised 12/82; accepted 12/82