SPELLING CORRECTION AND SYSTEMS PROGRAMMING

by

Howard L. Morgan

Technical Report
No. 69-31
February 1969

Department of Computer Science
Cornell University
Ithaca, New York 14850

SPELLING CORRECTION AND SYSTEMS PROGRAMMING

Howard L. Morgan*

Cornell University, Ithaca, New York

ABSTRACT: Several specialized techniques are shown to make the incorporation of spelling correction algorithms into compilers and operating systems efficient. These include the use of syntax and semantics information, restricted keyword and symbol table organizations, and consideration of only a limited class of spelling errors. The number of debugging runs per program has been cut down by using systems which perform spelling correction, saving both programmer and machine time.

KEY WORDS AND PHRASES: spelling correction, error correction, debugging, compilers, operating systems, diagnostics, error detection, misspelling, lexical analysis, systems programming.

CR CATEGORIES: 4.42 (debugging), 4.12 (compilers), 4.30 (supervisory systems;general), 3.79 (information retrieval; miscellaneous).

---

*Department of Operations Research and Department of Computer Science.

SPELLING CORRECTION AND SYSTEMS PROGRAMMING

Howard L. Morgan*

Cornell University, Ithaca, New York

Systems programs (language processors, operating systems, etc.) have become increasingly powerful and sophisticated yet, with a very few exceptions, do not assist the user by correcting many obvious spelling errors in the source input. While many errors in spelling are, of course, undetectable by the system since they violate neither syntax nor global semantics requirements, some spelling errors can be detected and corrected. The goal of the spelling correction techniques to be proposed can be roughly stated as achieving 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. For example, consider the following deck:

```
/JOB  2065  MORGAN, H.    10S   30P
/CUP6
     READ  ROWSUB, COLSUB, NOCOLS
     MATSUB = ROWSUB + COLSUB * NOCOLS
     WRITE MATSBU
     STPO
*DATA
3.0   4,  5
/ENDJOB
```

This fairly simple program, written in the CUPL language, reads three numbers and calculates a fourth number, MATSUB. Having been told that the program is in the CUPL language, most readers have assumed without hesitation that the second

---

*Department of Operations Research and Department of Computer Science.

card should probably read /CUPL , and, after a quick glance through the program, have also assumed (again correctly) that the last two program statements should read WRITE MATSUB and STOP , respectively.

Most operating systems would not make the first assumption and, even after a second pass at the machine brings in the CUPL compiler, few systems will do more than ignore the "STPO" statement and print messages at the end of the source listing to the effect, "SUSPICIOUS USAGE OF VARIABLE MATSBU", and a corresponding line for MATSUB . However, there do exist systems which would have detected and corrected all of these errors, and run the job as the user intended. This example, while contrived, is not atypical of the types of errors encountered in ordinary programs, and the systems make frequent and useful corrections.

Moreover, it is not particularly costly in machine time to detect and correct these errors. Efficiency is obtained in three ways:

1. A limited class of spelling errors is considered;
2. Syntax and global semantics information is used;
3. Special symbol table organizations are used.

PREVIOUS WORK

Previous work on spelling correction has been reported as far back as 1957 (Glantz [7]). Most of the work has focused on the general problem of recognizing misepelled words in a string of plain English text, given a dictionary of "correct" spellings. Blair [2] proposed a method of abbreviations which were specially constructed to retain the information content of the letters which made up the word. The first letter, for example, had more weight than subsequent letters, and consonants had more weight than vowels.

Damerau [4] proposed an algorithm which is incorporated into the algorithms presented in this paper, but was concerned with an indexing application. Alberga [1] presents a comparison study of many different spelling correction algorithms.

Two works have reported on the problem of correcting misspelled names. One, by Davidson [5], reported an algorithm similar to Blair's for use in retrieving misspelled airline passenger names. Another algorithm has been reported by Jackson [9] for use in correcting company names for a stock market data base application.

The only work on correcting spelling errors in the context of compilers and operating systems which this author is aware of was done by Freeman [6] for the CORC language at Cornell University. Freeman used a fairly sophisticated technique for estimating the probability that a suspiciously used variable or label name was in fact some other, properly used variable or label name. He evaluated a scoring function which made use of such information as the number of letters in the new word which matched corresponding ones in the dictionary word, the number of letters which matched after one or two character transpositions or substitutions, and even the number of letters which matched after taking into account the possibility of inadvertent use of the shift key on the keypunch. This technique correctly identified many misspellings, but was somewhat inefficient because of the large number of possibilities tried. (Of course, in the case of a core-resident load and go system such as CORC [3], the inefficiency was hardly noticed when compared against other, conventional systems.)

THE BASIC ALGORITHM

The spelling correction algorithm used here has a two stage structure (this structure seems common to almost all of the algorithms seen). If one calls the word whose spelling is in question the "entry word", and the words whose spelling is known the "dictionary words", one can present the structure as follows:

1st stage. Select a subset of the dictionary words which contains all those words of which the entry word may be a misspelling.

2nd stage. For each member of the above subset, apply a pairwise comparison algorithm to determine whether or not the entry word is a misspelling of the chosen member.

These two stages can operate as either coroutines or as subroutines, although the coroutine organization is probably more efficient. The problems of constructing an algorithm to handle the second stage are described in Damerau [4], and the algorithm which he presents for pairwise comparison is used as the second stage in the operating systems and compilers which this author has built. Approaches such as scoring functions, compressed representations of words, or numerical transformations of words are mainly directed with getting an efficient algorithm for the second stage.

Assuming that the second stage algorithm takes approximately constant time t_2 for each member of the subset chosen in the first stage, the time for the overall procedure will be mt_2, where m is the number of items in the subset. Significant reductions in either m or t_2 will result in a more efficient procedure from a time standpoint.

Several techniques are proposed which will result in the reduction of one or the other of these parameters.

LIMITED CLASS OF SPELLING ERRORS

Clearly, the types of misspellings which can occur in source text are many in number, and looking for all of them would be a difficult, if not impossible task. But, as Damerau [4] has noted:

> An inspection of those items rejected because of spelling errors showed that over 80 percent fell into one of four classes of single error — one letter was wrong, or one letter was missing, or an extra letter had been inserted, or two adjacent characters had been transposed.

This report has been confirmed by the author after an analysis of several hundred student programs written in the FORTRAN and CUPL languages at Cornell. Note also that all four of these types of errors are those which arise naturally from misuse of a keyboard input device (keypunch, teletype, etc.).

By limiting the class of spelling errors looked at to the categories described above, one can reduce both m and t_2. Since only single errors are considered, any dictionary words whose length differs by more than one from the entry word may be rejected immediately, thus reducing m. Also, the pairwise comparison algorithm is simpler than if multiple errors were being checked for, thus lowering t_2.

In compilers and operating systems, where most data input is from keyboard type devices, the above restrictions seem quite reasonable a priori, and have proven so in practice.

USE OF SYNTAX AND SEMANTICS INFORMATION

In job-control or high level programming languages, there is a well-defined syntax and semantics which must be followed if a program is to do what the programmer intended it to do. Knowing these syntax and semantics rules, the systems programmer can greatly reduce the size of the subset chosen in the first stage of the spelling correction algorithm (m), and can also reduce the number of times that the correction algorithm is invoked.

Most translators consist in part of a syntactic analysis pass followed by a semantics pass. Different types of spelling errors can be corrected during each of these passes, and the information which is needed for the translation process (be it for a job control or programming language) can be used to advantage in controlling the spelling correction algorithm.

During the syntactic analysis phase, the keywords of the statements must be recognized or the statement will presumably be ignored or called in error. Misspellings of keywords should be checked at this time. For user defined symbols, i.e. variables and labels, spelling correction can be postponed until the symbols are bound in the semantic phase.

Control of the spelling correction algorithm during syntactic analysis might work as follows: If a keyword is required by the syntax, and some other symbol appears, check this symbol against those keywords which would be legal in the current context. For example, in the program shown at the beginning of this article, the job control language requires a keyword to be the first word on a control card. Thus, when the /CUP6 card is read, and CUP6 is found to be unknown, this word is passed to the spelling correction algorithm, along with an indication that only the class of allowable first words need be examined. The stage one algorithm takes this into account when constructing the subset, and limits m to a manageable small number.

Similarly, the compiler writer can specify classes of keywords which need be considered by the stage one algorithm. When processing an IF statement, for example, only relational operators need be checked. In the CUPL compiler, the classes of keywords used are:

1. statement keywords, e.g., READ WRITE, ALLOCATE, etc.
2. relational operator, e.g., AND, OR, LE, GE, GT, etc.
3. function name, e.g., SIN, COS, SQRT, etc.
4. auxiliary, e.g., FOR, TO. BY, WHILE

If the spelling corrector cannot find a match in the desired class, no other classes are checked, since the statement is illegal.

During the semantics phase, the global usage requirements of the program are taken into account in order to control the spelling correction algorithm. If a variable or label name is suspiciously used (i.e. appearing only on the right hand side of assignment statements) the entry word is set as the suspicious variable, and the type of usage is passed to the spelling corrector (i.e. simple variable, array, label, subroutine name). The stage one algorithm can then ignore any dictionary words which are not of the same type. Thus, an array name will not be considered when looking for misspellings of a simple variable name. Empirical evidence indicates that it is inefficient to attempt correction of variable names with fewer than three characters.

To restate the idea of this section in a concise manner: Only those words whose misspellings could make sense in the context of syntax and semantics requirements of the programming language are tested against the entry word.

TABLE ORGANIZATION

The standard practice when designing a translator program is to specify a single global symbol table into which all keywords and user defined symbols are entered by means of some hashing function. Certain hashing functions are more useful than others when spelling correction is to be performed.

If only the class of single errors discussed above is considered, one can easily see that only those dictionary words whose first or second letter is the same as the first or second letter of the entry word need be included in the subset generated by the first stage of the spelling correction algorithm.
For example, if the entry word is STPO , the dictionary words that may be considered are:

1. Those beginning with S or T ;
2. Those whose second letter is S or T .

With this in mind, what type of hashing function will allow us to easily locate all members of one or the other of these groups? If the entry word is composed of the characters $a_1a_2a_3 \ldots a_n$, then a hashing function of the form

$$f_1(a_1) + f_2(a_2) + f_3(a_3, a_4, \ldots, a_n)$$

may permit us to quickly find all of the words beginning with or whose second letter is any given letter. If the ranges of the three functions are carefully chosen, the spelling correction algorithm can perform what is essentially a tree search of the entire hash table in order to find words with the necessary first or second letters.

If the range of f_3 is small, and the ranges of f_1 and f_2 are chosen so that, for example,

$f_1(B) - f_1(A) > f_2(Z) - f_2(A) + \text{range}\,[f_3]$, one can find all keywords whose first letter is A by a linear search starting at $f_1(A)$ and ending at $f_1(B)$. Of course, in practice, other considerations of hashing algorithms, such as percent of table full, and amount of scattering desired to reduce collisions, may dictate against constructing the hashing function in precisely this manner. However, one can always construct a chained list through all symbols with given first or second letters, in order to save time during spelling correction (at the expense of core).

A third possibility which should be considered when using hash tables is that of exhaustive search. Since hashed tables are not usually filled, some way of quickly getting all non-null entries should be made available.

SUMMARY

The ideas contained above have been used in designing spelling correction algorithms for the CUPL and DPL compilers and the Cornell Online Operating Systems on 360 computers. In use, the number of passes needed to get a program to run as the programmer intended has been reduced. In the case of CUPL, a simple FORTRAN-like language, an average of two runs is needed to get proper output, as opposed to more than five for the same programs written in FORTRAN.

An algorithm similar to that discussed above could be applied to FORTRAN, PL/I, and Job-Control Language scanners, with a view to correcting what Hamming [8] has said is one of the key failures of software people, namely, we do not use the computer to help us as much as we could.

REFERENCES

1. Alberga, Cyril N., "String Similarity and Misspellings," Comm. ACM 10, No. 5 (May 1967), 302-313.

2. Blair, Charles R., "A Program for Correcting Spelling Err Information and Control 3 (March 1960), 60-67.

3. Conway, R. W. and W. L. Maxwell, "CORC - The Cornell Computing Language," Comm. ACM 6 (June 1963), 317-321.

4. Damerau, F., "A Technique for Computer Detection and Correction of Spelling Errors," Comm. ACM 7 (March 1964), 171-176.

5. Davidson, L., "Retrieval of Misspelled Names in an Airlin Passenger Reservation System," Comm. ACM 5 (March 1962), 169-171.

6. Freeman, D. N., "Error Correction in CORC: The Cornell Computing Language," Ph.D. Thesis, Cornell University, Ithaca, New York, September 1963.

7. Glantz, H. T., "On the Recognition of Information with a Digital Computer," J. ACM 4 (1957), 178-188.

8. Hamming, R. W., "One Man's View of Computer Science," J. ACM 16, (January 1969), 3-12.

9. Jackson, M., "Mnemonics," Datamation 13 (April 1967), 26-

ors,"

es

-29.