Information Processing & Management Vol. 23, No. 5, pp. 477–494, 1987
Printed in Great Britain.

0306-4573/87 $3.00 + .00

A LOGICAL FRAMEWORK FOR THE CORRECTION OF SPELLING ERRORS IN ELECTRONIC DOCUMENTS

H. L. BERGHEL
Department of Computer Science, University of Arkansas, Fayetteville, AR 72701

(*Received* 22 *July* 1986; *accepted in final form* 16 *January* 1987)

Abstract—We propose a method for the correction of spelling errors found in electronic documents which is derived from a logical analysis of the problem. Specifically, set-theoretical definitions are given for similarity relations which describe certain properties which character strings may have in common. These definitions are then directly encoded into a PROLOG program. The advantages and disadvantages of this method are discussed, and some suggestions for further research are made. A detailed literature review is offered in order to place this method in perspective.

1. INTRODUCTION

In our view, a spelling checker worthy of the name must be capable of performing at least three operations: *document normalization, spelling verification* and *spelling correction.* The first operation is pragmatically motivated; unless the document is normalized, there is no simple way to identify the constituent words. The latter two operations are conceptually related though functionally distinct. Verification identifies candidate misspellings while correction suggests acceptable substitutions. In general, the objective of spelling checking is to "massage" an input electronic document into one which is free of spelling errors. The logic of the problem dictates that this must occur in ordered stages.

1.1 Document normalization

The first phase of spelling checking is *document normalization.* This entails, at least, the standardization of character encodings with regard to case, font, character set, etc., so that distinct word tokens of the same type can be recognized as such. To illustrate, the program must be capable of recognizing that "DOG" and "dog" are orthographically identical, despite typographical differences, for both are correctly spelled tokens of the same English word. While this procedure is nontrivial because the correction process must not destroy the original format of the text, it is amenable to standard string-manipulation procedures.

Second, it may be the case that normalization will require the removal of formatting symbols. This is particularly true of electronic documents created with modern word processing software with concurrent formatting, for there may be no unformatted copy of the document to work with. Some of the formatting symbols, the so-called "control characters," are nonprintable characters. These formatting symbols are by far the easiest to work with for their encodings are mutually exclusive with those of the text. More problematic is the set of printable formatting characters, for they may serve a variety of functions in the electronic document. To illustrate, hyphens which function as delimiters (as in the case of social security numbers and compound words) are essential, whereas those which signify word-breaks at lines' end are extraneous.

Third, grammatical and punctuation symbols must be handled in some fashion. Again, this operation must be accomplished with finesse, for the symbols may be used for a variety of purposes. For example, apostrophes which serve as "scare quotes" must be extracted, while those which indicate contractions would normally be preserved. In addition, characters which are used as word boundaries (periods, commas, spaces, etc.) must be stripped from the adjoining character strings before comparison.

Last, we may require some standardization for alternative spellings ("judgment" vs.

"judgement") in order to reduce number of dictionary entries. In such a case, one alternative might be normalized to another. This might be of value in handling both British and American spellings with one lexicon, for example. One might wish to extend the principle of standardized spelling even further by reducing word tokens to root forms and affixes.

These operations illustrate the sorts of activities that are involved in converting an electronic document into a *canonical form*. Once in this form, presumably all word-candidates are identified and the document is amenable to verification and correction. First, the *verification procedure* will identify those word tokens whose spelling cannot be verified. Next, a *correction procedure* will suggest a set of alternative spellings for each unverified token. In the ideal case, the correction would be largely automatic, where each misspelled word would be replaced with the intended, correctly spelled surrogate. However, this goal is unrealistic given that no computer can recover the writer's intentions. The best that we can hope for is an "educated guess" based upon the syntactic and semantic context.

1.2 Spelling verification

While the three-fold approach to spelling checking is widely accepted and employed in almost all computerized spelling programs, there is no consensus concerning the optimal strategies for their implementation. This is particularly true when it comes to the latter two stages.

For example, in the case of verification (the determination that there is in fact a spelling error), there are at least two radically different approaches which have currency in the literature. On the one hand, there are those who favor a *deterministic* approach. Exemplary techniques include *pattern matching* and *table look-up*. In this case, verification amounts to the logical intersection of a set of "target" words and a lexicon. (From now on, we will use the term "lexicon" rather than "dictionary" when referring to word lists without accompanying definitions.) The main advantage of this approach is that it is exact (assuming a lexicon of sufficient size). One major disadvantage is that it is difficult to do quickly without a primary-resident dictionary of considerable size. By consuming large amounts of primary storage, the pattern matching technique favors machines with larger internal memories.

The principles behind lexical searching are well understood, and the algorithms are highly refined. For a general discussion, see Knuth [21]. Several sublinear string search algorithms are available [9,22] which presumably have found their way into current software products. In addition, considerable attention has been given to the organization of the lexicon to optimize searching. This organization has been based on partial [18] and total [7] hashing, tree-structured domains [28,30], and segmented lists [26,40], to name but a few.

The second common approach to spelling verification is *probabilistic*. This is typified by *constituent analysis*. In this case, no attempt is made to match a target string with a lexical entry. Rather, algorithms are employed to associate with each target a "coefficient of peculiarity." Those targets which seem most peculiar are identified as likely candidates for misspellings. This type of analysis can be much faster than search routines, although they are obviously less precise.

The most common form of constituent analysis is "n-gram" or "string segment" analysis. An n-gram of a string $c_1 c_2 \ldots c_L$ is any segment of length $n < L$. Thus, the digrams of "ABCD" are "AB," "BC" and "CD"; while the trigrams are "ABC" and "BCD." In general, for a string of length L, there will be $L - n + 1$ n-grams. If we adopt the convention that each character should appear in exactly n n-grams, we pad the ends of the string with $n - 1$ spaces thereby creating $L + n - 1$ n-grams.

n-Gram analysis is reasonable because of the nonuniform distribution of possible n-grams in the lexicon. To illustrate, in the 10,000 word lexicon which we use in our lab, only 68% of the 26^2 possible digrams actually occurred. This figure shrank to 20% for trigrams, and 2% for tetragrams. Similar results are reported by Peterson [31], Thomas [41] and Zamora *et al.* [47]. This clearly suggests that for $n > 2$ the majority of n-grams are indicative of an unusual spelling or misspelling. Of course, the accuracy of n-gram analysis is proportional to the degree to which the character composition of the input text mirrors that of the lexicon in use. The received view seems to be that trigram analysis offers

an ideal balance between the imprecision of digram analysis and the prohibitive computational complexity of higher-order *n*-gram analysis [2,47].

Once the target word has been decomposed into its constituent *n*-grams, each *n*-gram is assigned a weight which reflects its frequency of occurrence in some corpus (presumably, the lexicon in use). In the simplest form of constituent analysis, these weights are then used directly in the construction of an overall index of peculiarity for the containing word. This is basically the technique used in the TYPO system [27].

Of course, there are many variations on this theme. Zamora *et al.* [47] extend the analysis by calculating an *n*-gram error probability which is the ratio of the frequency of occurrence of *n*-grams in misspellings to its overall frequency. By using error probabilities rather than simple frequency measures, it is possible to avoid flagging many correctly spelled, though orthographically unusual, words.

It should be remembered that *n*-gram analysis is a subclass of constituent analysis. There are also verification techniques based upon graphemic analysis [46] (a grapheme is taken to be an orthographical approximation of a phonetic representation). By incorporating phonetic information into the technique, there exists a facility for identifying misspellings which are a result of "spelling by sound" (e.g. "numonia" vs. "pneumonia").

In addition, there are constituent analyses based upon information-theoretic principles. An example of this approach appears in Pollock and Zamora [33]. In this case, "keys" are associated with each target. These keys are transformations of the target with some of the orthographical redundancy eliminated. They are based upon the general principle of data compaction which holds that it is possible to eliminate redundancy in data, without loss of meaning. In the case of Pollock and Zamora, the compaction results from the elimination of duplicate characters, while preserving the original orders of first occurrence. These sorts of compaction techniques have their modern origins in the SOUNDEX system [29]. While this type of constituent analysis is associated with spelling correction (see below), it is easy to imagine an extension to verification as well: one could construct a peculiarity index for the compacted representation.

In addition, it should be mentioned that there are also hybrid techniques which have features common to both look-up and constituent analyses. "Stripping" is such a technique. In this case, constituent analysis is used to identify and remove legitimate affixes from word tokens. Then a table look-up procedure is employed to determine whether the root of the word is correctly spelled. This allows the verifier to check a large number of variations of the same root word, which minimal lexicon. Stripping has been used in several spelling checkers, most notably the Stanford spelling checker, SPELL. For further details, see Peterson [31] and Turba [43].

This by no means exhausts the range of spelling verifiers, whether conceived or implemented. However, it does call attention to the general considerations which guide their design.

1.3 Spelling correction

The last stage of spelling checking is the attempt to correct the detected errors. As one might expect, this attempt may be based upon a wide variety of techniques. The simplest spelling corrector will be an interactive facility which allows the user to choose a substitute from a list of alternative words. Of course, the key to the effectiveness of this sort of corrector will be the procedure which creates the list. (We will return to this topic, below.) At the other extreme, correction would involve the actual substitution of the correct spelling of the intended word for its misspelled counterpart. As we mentioned above, this "fully automated" approach would require a significant (and unrealistic, given the state-of-the-art) level of machine intelligence. A corrector which uses grammatical information about the context of the misspelled target occupies a middle ground. This is the ultimate goal of the current project [3].

Spelling correction, unlike verification, must of necessity involve a lexicon, for the notion of "correctly spelled word" is defined in terms of membership in one or more lexical lists. Usually, whenever primary storage is at a premium (as in current microcomputer environments), several lexical lists are used. Typically, these lists are arranged in a hierar-

chical fashion. A small lexicon which consists of very common words is kept in primary memory. This is complemented with a much larger, master lexicon on secondary storage. In some cases, tertiary lists are also employed for application-specific (e.g. "professional dictionaries" used in medical and legal offices) and user-specific terminology.

Lexical lists can be organized in a variety of ways. The simplest form is a sequential list where words are stored alphabetically, just as they would appear in a dictionary. In order to increase efficiency, one would normally index this list so that the partitions of interest can be accessed directly. The two most common elementary indexing techniques are those which index by collating sequence and those which index by length. Many algorithms employ both of these indexing techniques in list organization.

The two other data structures used for indexing word lists are tree structures, either at the word or character level, and hashing, whether partial or complete. For extensive treatment of the search characteristics of these techniques, see Knuth [21]. Turba [43] provides detailed discussion of their use in the organization of lexicons for spelling checking.

We assume, then, that a spelling corrector will have access to one or more lexical lists, organized in such a way that efficient retrieval is possible. The objective of the corrector can be stated quite simply: find the subset of lexical entries which are "similar" to the misspelled targets. The trick is to operationalize the similarity relation in such a way that the resulting procedure is coextensive with the relation. These issues are generally subsumed under the topic of approximate string matching to which we now turn.

2. APPROXIMATE STRING MATCHING

In their recent survey article [19], Hall and Dowling present a lucid account of the difference in motivation between spelling verification and spelling correction. On their account, verification involves determining whether a target word stands in an *equivalence relation* with a word in the lexicon, where two words are said to be in an equivalence relation if they "are superficially different and can be substituted for each other in all contexts without making any difference in meaning" (p. 383). While there seems to be some circularity in this definition, for the notion of "superficially different" would normally be defined by means of the same criteria as "equivalent," it provides us with a reasonable, albeit imprecise, understanding of the principle of verification. In practice, equivalence relations are identified by means of *exact string matching* techniques (a simple search is such a technique).

Spelling correction, on the other hand, involves the association of a target word (presumably, one which is misspelled) with a set of words in the lexicon which satisfy a *similarity relation.* We will think of the set of words which are in this similarity relation with the target as a *similarity set.* The essence of spelling correction can now be seen to be a two-fold activity involving the application of appropriate similarity relations to create the desired similarity sets, for all misspelled words in the document. Again, in practice, similarity relations are implemented by means of *approximate string matching* procedures.

Formally, approximate string matching is a procedure which associates nonidentical character strings with one another on the basis of some criterion (or set of criteria) of similarity. That is, if S is a set of strings, we can think of the k-wise clustering of S into similarity sets $S_1, S_2, \ldots, S_k$ such that $(\forall S_i)(\forall s)(\forall s')(s,s' \in S_i \leftrightarrow s \sim s')$, where "$\sim$" denotes a similarity relation, and s, s' are distinct strings. There may, of course, be many identifiable similarity relations over S, each one of which would correspond to a distinct clustering. Further, a single string may be a member of several similarity sets for any given similarity relation.

2.1 Similarity relations

It remains for us to give a general description of the similarity relations denoted by the tilde in the previous section. Recall that these similarity relations will guide the construction of the similarity sets which will contain (at least) the correct spelling of the intended word. Obviously, the similarity relations themselves are the conceptually central components of the spelling corrector.

Faulk [15] provides a useful, straightforward categorization of similarity relations. In his view, such relations are of three main types. *Positional similarity* is a relation which refers to the degree to which matching characters in two strings are in the same position. *Ordinal similarity* refers to the degree to which the matching characters are in the same order. Finally, *material similarity* refers to the degree to which two strings consist of the same character tokens. Of course, Faulk's characterization deals exclusively with orthographical similarity relations. We will expand upon this topic shortly.

To illustrate this classification, we borrow upon the early work of Damerau [12]. In this classic paper on data errors, Damerau reports that four single errors account for the majority of typing mistakes. These four errors result from: accidental insertion of an unwanted character, accidental omission of a desired character, accidental substitution of one character for another, and accidental transposition of adjacent characters. According to Damerau, these four sources of mistakes, taken together, account for over 80% of all typing errors. This result was confirmed by Morgan [26] and Shaffer and Hardwick [38]. A later study by Pollock and Zamora [33] suggests that over 90% of all typing errors may involve a single mistake, though not necessarily those identified by Damerau.

Brief reflection will indicate that the Damerau conditions fall neatly into the classification scheme proposed by Faulk. Specifically, accidental substitution is an instance of positional similarity; transposition is a case of material similarity; and both accidental insertion and omission constitute examples of ordinal similarity. We now define these errors as specific similarity relations.

Let O, I, S and T be sets of character strings (word tokens) which are defined with respect to the Damerau conditions of omission, insertion, substitution and transposition, respectively. Let s be an arbitrary string and y be a character. We define O, I, S and T as sets of character strings similar to $c_1c_2c_3 \ldots c_{n-1}c_n$, such that

$$s \in I \leftrightarrow s = ((c_2c_3 \ldots c_n) \vee (c_1c_3 \ldots c_n) \vee \ldots \vee (c_1c_2 \ldots c_{n-1}))$$

$$s \in O \leftrightarrow (\exists y)(s = ((yc_1 \ldots c_n) \vee (c_1yc_2 \ldots c_n) \vee \ldots \vee (c_1c_2 \ldots c_ny)))$$

$$s \in S \leftrightarrow (\exists y)(s = ((yc_2c_3 \ldots c_n) \vee (c_1yc_3 \ldots c_n) \vee \ldots \vee (c_1c_2 \ldots c_{n-1}y)))$$

$$s \in T \leftrightarrow s = ((c_2c_1c_3 \ldots c_n) \vee (c_1c_3c_2 \ldots c_n) \vee \ldots \vee (c_1c_2 \ldots c_nc_{n-1})).$$

We make two observations concerning these biconditionals. First, the right-hand sides extensionally define the sets in question. Second, these extensional definitions are intensionally identical to the conditions defined by Damerau. In other words, the above biconditionals are precise set-theoretical specifications of the similarity relations implicit in the Damerau conditions. Shortly, we will describe a general procedure for directly implementing such descriptions in a spelling corrector. However, before we do this we will describe the conventional approaches toward similarity relations which motivate our own method.

2.2 Similarity measures

Recall from the above discussion that our intention in spelling correction is to define a similarity set for each misspelled target. We define this set in terms of one or more similarity relations. Now it remains for us to describe how we might actually operationalize these relations.

One common approach is to explicate the relation by some sort of *similarity measures.* As an example, the *n*-gram analysis techniques used for spelling verification are commonly used to create measures of similarity as well.

A number of similarity measures have appeared in the literature. Some of them provide a single type of similarity relation (i.e. in Faulk's sense). For example, Glantz [17] proposes a method which measures only positional similarity. While this method may be of use in restricted areas (e.g. OCR and literary comparison), it is generally felt [2,31] that positional similarity is too narrow to be of much use by itself in spelling correction.

Measures of material similarity have also been developed. Siderov [39] uses correla-

tion coefficients as a measure of character similarity. Unlike positional similarity, which is thought to be too narrow, material similarity is too broad for spelling correction (all anagrams are materially similar!). Thus, measures of material similarity are unlikely to be viable strategies for determining the extension of similarity sets.

Ordinal similarity measures are by far the most common. The early work in SOUNDEX [29] is the earliest example that we know of. Refinements have been discussed frequently in the literature [1,6], and have been used with some success in restricted applications [10,14]. Current research which deals with ordinal similarity measures includes the character-node trie approach of Muth and Tharp [28] and the similarity key method of Pollock and Zamora [33].

The essential weakness of single-relation measures is that they deal with only one aspect of string similarity. It is not surprising, therefore, that current interest has shifted to *agglomerative measures* which incorporate several similarity measures in their design.

2.3 Agglomerative measures of similarity

Following Hall and Dowling [19], we find it useful to distinguish between *metric* and *nonmetric* similarity measures. A typical metric similarity measure is a real-valued difference function, d, over character strings, which satisfies the conditions

(1) $d(s,s') \geq 0$
(2) $d(s,s') = 0 \leftrightarrow s = s'$
(3) $d(s,s') = d(s',s)$
(4) $d(s,s') + d(s',s'') \geq d(s,s'')$

for arbitrary character strings s, s', s''.

A popular similarity measure which is both agglomerative and metric is the so-called "Levenshtein Metric," named after the pioneer in coding theory who first suggested its use [23]. This measure has been applied to spelling correction by Wagner *et al.* [25,45]. We now describe this technique based upon the presentation in Hall and Dowling [19].

Let $d(i,j)$ be an agglomerative measure of similarity with respect to strings $s = c_1 c_2 \ldots c_i$ and $s' = c_1' c_2' \ldots c_j'$. We define it recursively as

$$d(0,0) = 0,$$

$$d(i,j) = \min \begin{cases} d(i,j-1) + 1, \\ d(i-1,j) + 1, \\ d(i-1,j-1) + v(c_i, c_j'), \\ d(i-2,j-2) + v((c_{i-1}, c_j') + v(c_i, c_{j-1}') + 1, \end{cases}$$

where

$$v(c_i, c_j') = 0 \leftrightarrow c_i = c_j' \qquad \text{and}$$

$$v(c_i, c_j') = 1 \leftrightarrow c_i \neq c_j'.$$

In this case, we extract a measure of similarity between two strings by creating a directed graph for all nodes (i,j), with horizontal and vertical edges weighted 1 (the penalty for a mismatch) and the weights of diagonal edges determined by v. Intuitively, since penalties are cumulative, the more dissimilar strings will have the longest, shortest paths. Since the difference measure, d, satisfies conditions (1)–(4) above, it qualifies as a legitimate metric.

Note here that the terms of the minimization function *approximate* the Damerau conditions set forth in Section 2.1. Assume that the shortest path weight is 1, for two strings whose lengths differ by 1. Further, assume that this path weight resulted from the mini-

mizing term $d(i-1,j)+1$. We would take this to mean that the shorter string is related to the longer by accidental omission of a character.

While there are other metric similarity measures, for example the probabilistic metric of Fu [16], the Levenshtein metric seems to be the dominant model. However, the greatest attention is now being given to nonmetric agglomerative similarity measures based upon n-gram analysis. We illustrate its use by describing the general structure of the procedure developed by Angell [2].

Let S and S' be sequences of n-grams corresponding to strings $c_1c_2 \ldots c_i$ and $c_1'c_2' \ldots c_j'$, respectively. We now determine the number of n-grams, k, which are common to S and S', by pairing them in order, as they appear in S and S', and discarding them after each match. We then stipulate that the measure of similarity is to be some value which is related to both k and the string lengths, say $2k/(i+j)$. This value may then be used to define the similarity sets for a string by simply requiring that it exceed a preestablished threshold. While this is just one particular application of n-gram analysis, variations on this theme abound [11,34,44,46,47].

In the above, we have tried to show how agglomerative measures of similarity are used to define similarity relations. In some cases, for example the Levenshtein metric, the connection between the relations and the measures is more obvious than in others (e.g. n-gram analysis). However, in any case, the use of similarity measures suffers from two defects: first, the measures are always approximations; second, they are of no immediate use in identifying the similarity sets required for spelling correction. To elaborate, while they can be of some value in determining whether two strings are similar, it is not at all clear that they would be useful in extracting the similar strings from a lexical list in the first place. One could, of course, employ the similarity measures as a basis for organizing the list itself (cf. [36]), but this strikes us as undesirable for it destroys the alphabetical ordering of the lexicon.

3. THE ACCURACY OF SIMILARITY MEASURES

As we saw in the last section, spelling correction requires that we have some procedure which identifies the members of the similarity sets for our misspelled tokens. What bothers us about the conventional approaches is that they use *measures* of similarity to define the actual similarity relations. It is our objective to develop a spelling corrector which directly operationalizes the similarity relations.

Our motivation comes from the observed inaccuracy of conventional approaches. This "inaccuracy" can be formally stated in terms of the familiar evaluation measures of information retrieval theory. Assume that there are j words in a lexicon of size s which are known to be similar to a target word. Let n be the size of a similarity set actually produced. Further, let $k \leq n$ be the number of members of the similarity set which actually do stand in the similarity relation to the token. We now define the following evaluation parameters after Salton [37]:

$$\text{recall} = k/j$$

$$\text{precision} = k/n$$

$$\text{fallout} = (n-k)/(s-j).$$

Our objective is to find a method of approximate string matching which has complete recall, 100% precision and zero fallout. That is, we are looking for a procedure, P_i, which will enumerate all and only those strings which are in the ith similarity relation to a target. Assume that we have *a priori* knowledge of the similarity set, S_i, which corresponds to a certain relation with respect to some word token. We then require that $(\forall x)(P_i \Rightarrow x \leftrightarrow x \in S_i)$. In other words, we require a procedure which is coextensive with the actual similarity relation.

One can best appreciate this goal when one refers to the descriptions of the perfor-

mance of conventional approaches in the literature. For example, Pollock and Zamora [33] report that only 71% of the misspelled word targets were corrected by their method; 11.54% were miscorrected, and 17.45% were uncorrected. Yannakoudakis and Fawthrop [46] reveal that greater than 25% of the target words caused their algorithm to fail completely. Even when the algorithm worked, it was only able to suggest the correct spelling of a word token which was corrupted by a Damerau error 90% of the time. These results seem to be typical of spelling error detectors and correctors based upon measures of similarity [47]. Although Muth and Tharp [28] reported greater effectiveness, the practicality of their approach seems to be in doubt [13].

4. APPROXIMATE STRING MATCHING AS THEOREM PROVING

We propose a new approach to spelling correction which overcomes the inaccuracies of similarity measures. In our method, we use standard logic programming techniques to directly implement the desired similarity relations. We do this by creating two sets of axioms, and then using an inference engine to derive counterexamples to the claim that there is no word which is similar to a misspelled token.

The first axiom set defines the lexicon. Consider a lexicon, L, which consists of predicative expressions of the form $W(x)$, where x is a string variable and the predicate is interpreted as "x is a word." We will assume that there be one such *lexical axiom* for each correctly spelled word of interest.

The second axiom set is a set of logical procedures $P = \{P_1, P_2, \ldots\}$ which will define the similarity relations in question. Our intention is to directly represent the similarity relations in terms of these *procedural axioms*, so that a procedure can be found which finds similarity sets without approximation.

We make some general observations concerning these axioms. First, we note that all correctly spelled words in the lexicon are logical consequences of the lexical axioms. Another way of putting this is that if x is a word (i.e. if $W(x)$ is one of the lexical axioms), then "$-$W(x)" is inconsistent with the axioms.

Second, if the procedural axioms, P, are complete and consistent with respect to the underlying relations, we can say that for all strings, s, s' $((W(s) \;\&\; ((W(s) \cup P) \vdash s')) \leftrightarrow s \sim s')$. That is, we can show that a string is similar to a target if and only if that string is logically implied by the axioms. Eventually, we will create procedural axioms which are coextensive with the set-theoretical definitions of the Damerau relations which we gave in Section 2.1. But first we will describe a general procedure for implementing similarity relations according to Faulk's classification.

4.1 Defining the axioms in PROLOG

While a detailed discussion of PROLOG is beyond the scope of this paper, a few general remarks are in order. First, PROLOG is a logic-based language which contains a built-in inference mechanism based upon the resolution principle defined by Robinson [35]. For our purposes, the important ingredient of this mechanism is the principle of *reductio ad absurdum* whose tautological counterpart can be expressed as $((A \rightarrow (B \;\&\; -B)) \rightarrow -A$. In semantic terms, this says that any formula which implies a contradiction must be false.

The resolution principle is defined for Horn clauses [20]. Horn clauses are elementary disjunctions with at most one unnegated atomic formula. Following convention, we will call Horn clauses with one unnegated formula *headed*, and those without, *headless*. We note that the truth-functional equivalence $B \vee -A_1 \vee \ldots \vee -A_n \leftrightarrow (B \leftarrow (A_1 \;\&\; \ldots \;\&\; A_n))$ means that we can represent any Horn clause as a conditional. In the notation of PROLOG, the "$\leftarrow$" is represented as ":$-$" and is referred to as the "neck" of the clause. Further, the conjunction is denoted by the comma.

Finally, PROLOG makes available an infinite number of functors, one of which is the list functor. For our purposes, we will represent lists with bracket-notation. In this way, we may think of the structure "$[e_1, \ldots, e_n]$" as an ordered sequence of n elements. We manipulate lists by splitting them into a head and a tail. The head, denoted by the "X" in the structure "$[X|Y]$," is the first element in the list, while the tail, "Y," is the remaining

sublist. By selective use of the list structure, we will be able to account for all three categories of similarity relations, as described by Faulk.

4.2 PROLOG schema for general similarity relations

Let $A = \{a_1, \ldots, a_k\}$ be an alphabet and A^* be a set of strings over A. Let $u, v, w, x \in A^*$. We define a string membership relation, $\hat{\in}$, in the following way: $v \hat{\in} w \leftrightarrow w = uvx$, for some $u, x \in w$. Further, we define a positional string membership relation, $\hat{\in}_k$, as follows: $v \hat{\in}_k w \leftrightarrow w = uvx \;\&\; |u| = k - 1$.

Now, we define three string difference coefficients, d_m, d_p, d_o, for material, positional and ordinal similarity, respectively. For each $a \in A$ and $w \in A^*$, let $n_a(w)$ be the number of occurrences of a in w (i.e. the number of instances in which $a \hat{\in} w$ is satisfied). Let f_m: $A^*XA^* \to N^k$ be a mapping from pairs of words onto k-tuples of integers. We define this function as follows:

$$f_m(v, w) = \langle |n_{a1}(v) - n_{a1}(w)|, \ldots, |n_{ak}(v) - n_{ak}(w)| \rangle.$$

We then define the material difference coefficient to be the result of summing the coefficients of the vector. That is,

$$d_m(v, w) = \sum_{i=1}^{k} (|n_{ai}(v) - n_{ai}(w)|).$$

We observe that $d_m(v, w)$ is in effect a count of the number of times that the logical expression $-(c \hat{\in} v \leftrightarrow c \hat{\in} w)$ is satisfied with respect to strings v, w for any character token, c. We make this observation because we feel that it is easier to see the correlation between the PROLOG clause sets and this logical expression, than it is for the clauses and the mathematical description.

Next, given $v, w \in A^*$. For non-null v, w, let $v = i(v)v'$ and $w = i(w)w'$, where $i(x)$ is the initial character token of x. We define the positional difference coefficient, d_p: $A^*XA^* \to N$, recursively:

$$d_p(v,0) = |v| = d_p(0, v)$$

$$d_p(v, w) = \begin{cases} d_p(v', w') & \text{if } i(v) = i(w) \\ 1 + d_p(v', w') & \text{if } i(v) \neq i(w). \end{cases}$$

Similarly, we note that $d_p(v, w)$ is a count of the number of times that the expression $-(c \hat{\in}_k v \leftrightarrow c \hat{\in}_k w)$ is satisfied.

Finally, let A^n be strings of length n over A. If $x \in A^n$ and $w \in A^*$ then let $n_x(w)$ denote the number of times that x appears as a segment in w. We define the ordinal difference coefficient by the function, d_o^n: $A^*XA^* \to N$, such that

$$d_o^n(v, w) = \sum_{x \in A^n} (|n_k(v) - n_k(w)|).$$

In this case, the logical representation would involve the use of another relation for "string-segment membership." We omit this expression in the interest of economy.

We now turn to the PROLOG realization of these functions. It is our objective to illustrate a general approach to the types of relations described by Faulk. Once this general approach is given, we will implement more specific similarity relations which are actually of use in spelling correction.

First, we observe that the string membership relation, $\hat{\in}$, can be explicated in terms of the PROLOG structure, "[_|_]." That is to say, if an element is in a list (in this case, a string) it is either the head of the list or in the tail. By defining string membership recursively, we can locate the element as the head of some sublist. This is accomplished by

means of the "string_member" predicate, below. In order to determine the number of mismatches, one augments the clause set with a "delete_token" predicate so that the same character token is not counted twice.

If the relation $\hat{\in}_k$ is of interest, we simply ensure that the heads of the lists (i.e. current character tokens of the strings) are always in the same respective position. This is easier to do, for it only involves a comparison of heads of strings (rather than the head of one string with an entire string). Notice that the more general membership predicate is unnecessary in the clause set for positional similarity.

Finally, we provide a general procedure for determining the degree to which two strings have matched characters in the same order. Inasmuch as ordinal similarity has to be determined with regard to string segments (ordinal similarity defined with regard to entire strings is positional similarity), this procedure is essentially an *n*-gram analysis. In general, *n*-grams are defined recursively as *n* consecutive heads of segments. For present purposes, we restrict our attention to trigrams.

We now offer illustrative procedures for the PROLOG realization of the three categories of string similarity defined by Faulk.

```
                    {MATERIAL SIMILARITY}

degree_of_m_similarity(STRING1,STRING2,COEFFICIENT):-
     ((string_member(TOKEN,STRING1),
       string_member (TOKEN, STRING2),
       not TOKEN = [],
       delete_token(TOKEN,STRING1,S1_SHORT),
       delete_token(TOKEN,STRING2,S2_SHORT),!,
       degree_of_m_similarity(S1_SHORT,S2_SHORT,N); N = 0));
          N = -1),
       COEFFICIENT is N+1.

string_member(X,[X|_]).
string_member(X,[_|Y]):-string_member(X,Y).

delete_token(X,[X|T],T).
delete_token(X,[_|Y],R):-delete_token(X,Y,R).

                    {POSITIONAL SIMILARITY}

degree_of_p_similarity([],[],0).
degree_of_p_similarity(STRING1,STRING2,COEFFICIENT):-
          STRING1 = [TOKEN|TAIL1],STRING2 = [TOKEN|TAIL2],
     degree_of_p_similarity(TAIL1,TAIL2,N),
     COEFFICIENT is N+1.

degree_of_p_similarity(STRING1,STRING2,COEFFICIENT):-
     STRING1 = [TOKEN1|TAIL1],
     STRING2 = [TOKEN2|TAIL2],
     not TOKEN1 = TOKEN2,
     degree_of_p_similarity(TAIL1,TAIL2,COEFFICIENT).

                    {ORDINAL SIMILARITY}

test:-    findall(X,trigram(STRING1,X),SEGMENT_SET1),
          findall(X,trigram(STRING2,X),SEGMENT_SET2),!,
          degree_of_o_similarity(SEGMENT_SET1,SEGMENT_SET2,COEFFI-
          CIENT).
```

```
degree_of_o_similarity(SEGMENT_SET1,SEGMENT_SET2,COEFFICIENT):-
        ((segment_set_member(SEGMENT,SEGMENT_SET1),
          segment_set_member(SEGMENT,SEGMENT_SET2),
          not SEGMENT = [],
          delete_segment(SEGMENT,SEGMENT_SET1,SHORT_SET1),
          delete_segment(SEGMENT,SEGMENT_SET2,SHORT_SET2),!,
          (degree_of_o_similarity(SHORT_SET1,SHORT_SET2,N); N = 0));
              N = -1),
          COEFFICIENT is N+1.

trigram([X],[]).
trigram([X,Y],[]).
trigram([X|[Y|[Z|T]]],[X,Y,Z]).
trigram([H|T],Tt):-trigram(T,Tt), not Tt = [].

segment_set_member(X,[X|_]).
segment_set_member(X,[_|Y]):-segment_set_member(X,Y).

delete_segment(X,[X|T],T).
delete_segment(X,[_|Y],R):-delete_segment(X,Y,R).
```

4.3 PROLOG implementation of Damerau relations

In the previous section, we saw how one might approach the categories of similarity relations described by Faulk. We now apply these techniques to the specific similarity relations suggested by Damerau.

We begin with the set theoretical descriptions presented in Section 2.1. We define the extension of the similarity sets *O*, *I*, *S* and *T* by means of PROLOG clauses which are defined with respect to a list representation of the character string, $c_1 c_2 \ldots c_n$. For example, the extension of *O* is found to be defined by

```
is_an_element_of_o(X):-                    {OMISSION}
        add_character(X,Y),
        word(Y).
```

where

```
add_character(X,[Y|X]).
add_character([U|V],[U|W]):-
        add_character(V,W).
```

This clause set is read in the following way: a character string, *X*, is an element of the set in question (the similarity set for string $c_1 c_2 \ldots c_n$ which corresponds to the similarity relation of omission) if the result of adding a character to the string is itself a word. "Adding a character" is then defined by using the PROLOG list notation. In the first clause for the predicate, "add_character," the uninstantiated variable, *Y*, represents the character-slot in the first position which will take on the added character. If this clause fails, i.e. if the cause of the spelling error was not a missing character in the first position, the second clause inserts a variable in the 2nd through *n*th position, one by one, by systematically inserting the uninstantiated variable as the head of tails of lists.

As the above example illustrates, the PROLOG code is trivial once one has the set-theoretical definition of the similarity relation and a general understanding of the underlying logic behind tests for ordinal similarity. Since the Damerau condition is a special case of a test for ordinal similarity, where the only segments of interest are those to the left and right of the uninstantiated variable, the code fragment above is much simpler than that for the general case.

We now present PROLOG clauses for the remaining Damerau conditions without discussion.

```
is_an_element_of_i(X):-                          {INSERTION}
      delete_character(X,Y),
      word(Y).
 delete_character([X|Y],Y).
 delete_character([X|U],[X|V]):-
    delete_character(U,V)

is_an_element_of_s(X):-                          {SUBSTITUTION}
      replace_character(X,Y),
      word(Y).
replace_character([X|Y],[U|Y]).
replace_character([X|V],[X|W]:-
 replace_character(V,W).

is_an_element_of_t(X):-                          {TRANSPOSITION}
    transposes_to(X,Y),
    word(Y).
 transposes_to([],[]).
 transposes_to([X|Y],[X|Z]):-
    transposes_to(Y,Z).
 transposes_to([X|[W|Y]],[W|[X|Y]]).
```

The actual spelling correction is a byproduct of the resolution/unification mechanism of PROLOG. Remember that the clause sets beginning with "is_an_element_of" are the procedural axioms. If it can be determined that the spelling of a word, *w*, has not been verified, we query the data base with "*W*(*w*)?" The inference engine of PROLOG then attempts to find a contradiction for the negated fact, "−W(w)." However, the procedural axioms "massage" *w* into a variety of forms (related to *w* by the Damerau relations) and then test each of these forms against the lexicon. If a contradiction is found, this implies that at least one of the massaged forms is a word. More specifically, the current substitution instances for the characters in the word token will define the correctly spelled word(s). In the purest sense, word tokens corrupted by the sort of mistakes described by Damerau are theorems of the spelling correction system.

4.3 Extending the method

It should be clear from the foregoing that this approach to spelling correction is extendible to include any similarity relation which can be expressed through Horn clauses (which is, for all practical purposes, the entire set of similarity relations). For example, were we to be interested in correcting target words which were altered by unwanted repetitions of characters (which is common on keyboards with "repeat" keys), we could easily define the similarity relation by means of a new set. Let R be a set of strings similar to $c_1 c_2 \ldots c_n$, except that all triple repetitions are reduced to a single character. That is,

$$\begin{aligned} x \in R \leftrightarrow\ & (x = (c_3 \ldots c_n) \leftarrow (c_1 = c_2 = c_3)) \\ & \vee\ (x = (c_1 c_4 \ldots c_n) \leftarrow (c_2 = c_3 = c_4)) \\ & \vee\ \ldots . \end{aligned}$$

Further, were we to try to incorporate misspellings due to homonymous morphemes, we might define a set, H, such that

$$x \in H \leftrightarrow (x = (pc_3 \ldots c_n) \leftarrow f(c_1c_2) = p)$$
$$\vee\ (x = (c_1pc_4 \ldots c_n) \leftarrow f(c_2c_3) = p)$$
$$\vee\ \ldots,$$

where f would be a function which mapped one set of character strings (in this case, of length 2) onto another set (in this case, a single character) which is pronounced the same (e.g. "PH" onto "F" as in the incorrect spelling, "Philipinos"). In addition, multiple errors can be easily detected by nesting the similarity relations, rather than treating them independently. If affix-stripping were found to be useful, incorporation would amount to the addition of a trivial list-manipulation operation, and so forth.

As a specific illustration, we consider the enhancement of the program to include tests for two extra characters, two missing characters and two characters transposed around a third (e.g. "narutal" for "natural"). These have been identified by Peterson[32] as the next most common typing errors after Damerau-type errors. We illustrate the simplicity of their inclusion by juxtapositioning them with the related Damerau tests for insertion, omission and transposition (minus those portions of the clause sets which are unaffected by the change). To wit,

Insertion

```
is_an_element_of_i(X):-
      delete_character(X,Y),
      word(Y).
```

Double Insertion

```
is_an_element_of_di(X):-
      delete character(X,Y),
      delete character(Y,Z),
      word(Z).
```

Omission

```
is_an_element_of_o(X):-
      add_character(X,Y),
      word(Y).
```

Double Omission

```
is_an_element_of_do(X):-
      add_character(X,Y),
      add_character(Y,Z),
      word(Z).
```

Transposition

```
is_an_element_of_t(X):-
      transposes_to(X,Y),
      word(Y).

transposes_to([],[]).
transposes_to([X|Y],[X|Z]):-
      transposes_to(Y,Z).
transposes_to([X|[W|Y]],[W|[X|Y]])
```

Transposition Around a Third

```
is_an_element_of_tat(X):-
      tat_transposes_to(X,Y),
      word(Y).

tat_transposes_to([],[]).
tat_transposes_to([X|Y],[X|Z]):-
      tat_transposes_to(Y,Z).
tat_transposes_to([X|[M|[T1|T2]]],
                  [T1|[M|[X|T2]]]).
```

We note that the enhancements represent simple and intuitive modifications of the original tests, just as we would expect.

The variations are endless. The only general restriction is that the similarity relation has to be expressible in first order logic. Further, we note that the PROLOG code which follows trivially from the set-theoretical description is more concise, for its recursive procedures significantly simplify the iterative definitions.

The flexibility of the current approach applies to constraints on the search as well. At the highest level, search is controlled by simply adding, deleting and permuting the clause sets. Beneath that level, one can build any number of logical constraints into the clauses themselves. To illustrate, one might wish to encode "rules of thumb" into the program.

We have in mind such things as "i before e, except after c." For whatever its limitations, this rule of thumb is expressed in a logical form, and is easily encoded into the program. These sorts of "peremptory rules" might increase the efficiency of the correction procedure by precluding time-consuming searches for more general sorts of errors. That is, in the general case, it may be faster to preprocess the target and make a second attempt at verification than to attempt correction after the first verification failure. We would expect that this type of control structure would be primarily of use in the correction of "errors of ignorance" (vs. typing errors).

However, the most interesting type of extension is one which includes grammatical information as well. In particular, the logic-based approach to spelling correction is consistent with the definite clause grammar model for natural language parsing. As we have suggested elsewhere [3], a simple extension of the lexical axioms, to accommodate the grammatical features of words, can be used concurrently by both a spelling corrector and transformational parser (see also, [5]).

In short, the logic-based approach seems ideally suited to the types of operations which we might require of a spelling corrector. In fact, it is easy to demonstrate that even the rival agglomerative similarity measures can be expressed within this framework. Notice that the iterative definition of the Levenshtein distance metric can be replaced with the following recursive procedure:

```
d([],[],0).
d([X|T1],[X|T2],D):- d(T1,T2,D).
d([X|T1],[Y|T2],D):- X≠Y,d(T1,T2,Td), D is Td+1.
d([X|T],Y,D):- d(T,Y,Td), D is Td+1.
d(Y,[X|T],D):- d(Y,T,Td), D is Td+1.
d([H1|[H2|T1]],[H2|[H1|T2]],D):- d(T1,T2,Td), D is Td+1.
```

Where the first clause is the boundary condition, the second clause is the string-processing procedure, and the remaining clauses define the minimizing terms for substitution, omission, insertion and transposition, respectively. It almost goes without saying that our method can accommodate the *n*-gram analysis described in Section 2.3, for *n*-gram analysis is the paradigm case of Faulk's ordinal similarity. To wit,

```
t:-
    string_length(C1,L1),
    string_length(C2,L2),
    findall(T,find_trigram(C1,T),B1),
    findall(T,find_trigram(C2,T),B2),
    count(B1,B2,K),
    similarity_measure is (2*K)/(L1+L2),

    find_trigram([X|[Y|[Z|T]]],[X,Y,Z]).
    find_trigram([H|T],Tt):-find_trigram(T,Tt).

    count(L1,L2,N):-((member(M,L1),
        member(M,L2),
        not M = [],
        delete(M,L1,L11),
        delete(M,L2,L12)),!,
        ((count(L11,L12,Nr);Nr = 0); Nr = -1),
        N is Nr + 1.

    member(X,[X|_]).
    member(X,[_|Y]):-member(X,Y).

    delete(X,[X|T],T).
    delete(X,[_|Y],R):-delete(X,Y,R).
```

The capability of easily expressing the underlying logic of agglomerative measures of similarity gives some idea of the expressive power of the logic-based approach to spelling correction. We submit that this sort of versatility is unobtainable within the framework of the conventional approaches.

4.4 Efficiency of the method

We think that it is obvious from the above that we have achieved our objective of complete recall, 100% precision and zero fallout, with respect to the similarity relations in use. The reason for this is that the procedures are direct translations of the actual similarity relations. No measures or approximations of similarity are needed. Further, the procedures are easily generalized to other sorts of relations regardless of the source of error (e.g. typing, ignorance of spelling conventions). As we have shown above, this is a nearly trivial undertaking when the similarity relations fall into one of the categories defined by Faulk. We actually employ in our lab a prototype spelling corrector like the one outlined above. For a more complete description of an early version of the program, see [4].

While the ultimate efficiency of our method cannot be accurately predicted at this writing, some factors are known. For one, since the tests for similarity are always with respect to the lexicon, the search space is never larger than it would be with a brute force approach. Further, the number of required tests or comparisons will usually be less. For example, a generate-and-test strategy for the Damerau conditions of accidental substitution and omission would involve $n(k-1)$ and $k(n+1)$ tests, respectively, for a k-letter alphabet on strings of length n. These values are reduced to n and $n+1$, respectively, by our method. The reason for this is that the notion of "test" is inextricably linked to lexical entry. There is simply no opportunity to test any string which is not already a word due to the way that unification works.

In terms of the actual number of lexical comparisons which need to be made, the logic programming approach can be shown to be quite parsimonious. However, the case is not as clear with regard to searching. Typically, PROLOG clause sets are indexed by predicate name and arity. Given this indexing scheme, the search characteristics should be roughly the same as with conventional algorithms defined over length-segmented lists hashed to segment boundaries and searched sequentially within segments. This is consistent with our empirical data. When we compared PROLOG and PASCAL programs which checked sixteen misspellings for Damerau errors against a 474 word lexicon on an IBM PC/AT, the PASCAL program took 29.2 seconds and the PROLOG program ran in 7.3 seconds. However, the data structure used in the PASCAL program was a length-segmented list! This result only shows that our logic-based approach to spelling correction is competitive with a high-level approach, so long as the high-level approach is burdened with the same inefficient search strategy.

To provide some estimate of the degree of inefficiency, we processed an electronic document with four Damerau-related spelling errors. Each of the following similarity sets were generated in approximately 3 seconds when running a compiled version of our program on an IBM/PC AT with a dictionary of 4000 words:

target	*similarity set*
hte ⇒	{the,hate,he,ate,hoe,hue}
bal ⇒	{ball,bald,balk,al,pal,bad,bag,bar,bay}
warr ⇒	{war,ward,warm,warn,wars}
rwd ⇒	{red,rid,rod}.

On this basis, an electronic document of 5000 words with 10% errors would require approximately 25 minutes to correct. However, there are some mitigating circumstances.

First, the next wave of PROLOG compilers will include more efficient search strategies. With sophisticated hashing schemes or the use of multiway search trees, PROLOG searches will accelerate (there is no reason, in principle, that PROLOG cannot use the same

range of data structures as a high-level language). In fact, at least one current product supports B-trees. We are currently exploring these avenues.

Second, PROLOG is not well suited for serial, von Neumann architectures. Our host system is an IBM PC, which delivers a maximum of 10^3 logical inferences per second (LIPS). This is between two and three orders of magnitude slower than the rate which is within current technological limits [42] and seven orders of magnitude slower than the fifth-generation machines planned by ICOT. An increase in performance of a few orders of magnitude yields run times which are competetive with the conventional approaches. Thus, if we are not obsessed with short-term efficiency, the effectiveness of the logical approach justifies continued interest. (As an aside, we mention that it takes approximately $20n$ logical inferences to apply all four Damerau conditions to a string of length n, so the execution time of a similar program can be quickly approximated if one knows the speed of the host PROLOG system in LIPS.)

Third, we observe that most current PROLOG environments are poorly suited for integration with conventional, high-level languages. In our opinion, traditional deterministic document normalization and verification procedures are entirely adequate. While these tasks can be easily handled in PROLOG as well, it is pointless to do so. The best of both worlds can be achieved when one can readily integrate the PROLOG-based corrector with a conventional verifier. In this case, the conventional, procedural approach handles most of the work (over 80% of the word tokens, by some estimates [8,24]) while the PROLOG portion handles the more difficult problems of correction. This lack of integration is more a short-term nuisance than a long-term difficulty.

5. CONCLUSIONS

What we have described above is a technique for spelling correction which directly encodes the set-theoretical definitions of a set of similarity relations into a PROLOG program. This program then returns the similarity sets in question to the user in the form of a list of alternative spellings. The actual behavior of our program is much like that of any other interactive spelling checker.

We feel that there are several advantages to the logic-based approach which makes it worthy of further study. First and foremost, the method is exact. This is a consequence of the logical, top-down design. Related to this is the fact that the approach allows the designer to solve the problems of spelling correction (or spelling checking, for that matter) at the conceptual rather than the procedural level. Given a precise description of the problem, either in terms of set theory, iterative or recursive definitions, the code results from a straightforward translation.

Further, a logic-based method avoids the necessity of *ad hoc* procedures common to the agglomerative measures of similarity. As an illustration, consider that the complexity of converting the single Damerau condition of transposition into a distance metric prompted a separate article [25]. We maintain that this is a direct result of what we see as an unintuitive and awkward representation of the problem. Similarly, the more advanced, nonmetric, n-gram analyses are built upon elaborate procedures for calculating frequency analyses, error probabilities and the like for n-grams, none of which seem *prima facie* related to the topic of spelling errors.

We suspect that the environment in which a spelling checker is used will have an influence upon the particular tests that are found to be useful. We would expect that accomplished spellers could rely heavily on the verification procedure alone, since most of their errors are likely to be typographical. Poor spellers, on the other hand, are likely to expect much more of a spelling checker than knowledge that an error occurred. A significant advantage of the method under discussion is that it easily admits extensions and modifications. In this regard, it is quite unlike agglomerative measures. It is not at all obvious to us how one would extend distance metrics and n-gram analyses to cover a broad enough range of spelling errors that the resulting product would be relevant to a wide variety of applications. Further, the more rigorous the spelling checking procedure, the more natural it becomes to exploit the inherent AND-parallelism in PROLOG.

In addition, we note that certain spelling errors are impossible to correct when isolated from their grammatical context. The paradigm case of such errors are targets which have undergone "complete corruption." This occurs when an intended spelling has been corrupted to the point that it is a legitimate spelling of another word (e.g. the noun, "wand," corrupted to the verb, "want"; the preposition, "to," to the adjective, "too"). These sorts of errors can only be detected, and corrected, if the spelling checking is done in conjunction with at least a partial parse of the containing sentence. We see no way to generalize conventional spelling checkers beyond the orthographical level. Of course, there are even more serious forms of complete corruption which cannot be detected without an understanding of the sentence (i.e. when a word is corrupted into another word with similar grammatical properties, as in "wonder" → "wander"). We shall defer these topics to another forum.

Thus, we have in the present effort an attempt to provide a spelling correction facility which has the characteristics which we feel will be required in the future. What is interesting to us is not the particular details involved in developing a PROLOG-based system, for we would be satisfied with an equipotent system in any language environment, but rather that the solution follows directly from the logic of the problem. Even if the PROLOG approach fails to pass the test of time, the lessons which we can learn from its design should serve us well in providing a logical framework for products to follow.

Acknowledgements—We wish to thank M. Kunze, S. Margolis, R. Morein, G. Nagy, S. Seth, P. Smith, E. Traudt and the anonymous referees of *Information Processing and Management* for their invaluable comments and suggestions.

REFERENCES

1. Alberga, C. String similarity and misspellings. Communications of the ACM, 10: 302–313; 1967.
2. Angell, R.; Freund, G.; Willett, P. Automatic spelling correction using a trigram similarity measure. Information Processing and Management, 19: 255–261; 1983.
3. Berghel, H. Extending the capabilities of word processing software through Horn clause lexical databases. Proceedings of NCC-86: 251–257; 1986.
4. Berghel, H.; Traudt, E. Spelling verification in Prolog. SIGPLAN Notices, 21: 12–27; 1986.
5. Berghel, H.; Weeks, J. On implementing elementary movement transformations with Definite Clause Grammars." In: Proceedings of the fifth Phoenix conference on computers and communications. IEEE Computer Society; 1986: 366–370.
6. Blair, C. A program for correcting spelling errors. Information and Control, 3: 60–67; 1960.
7. Bloom, B. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13: 422–426; 1970.
8. Bourne, C. Frequency and impact of spelling errors in bibliographic data bases. Information Processing and Management, 13: 1–12; 1977.
9. Boyer, R.; Moore, J. A fast string searching algorithm. Communications of the ACM, 20: 762–772; 1977.
10. Bryant, J.; Fenlon, S. The design and implementation of an on-line index. Referenced in [15].
11. Cornew, R. A statistical method of spelling correction. Information and Control, 12: 79–93; 1968.
12. Damerau, F. A technique for computer detection and correction of spelling errors. Communications of the ACM, 7: 171–176; 1964.
13. Damerau, F. Review of [28], Computing Reviews, June: 231; 1978.
14. Davidson, L. Retrieval of misspelled names in an airlines passenger record system. Communications of the ACM, 5: 169–171; 1962.
15. Faulk, R. An inductive approach to language translation. Communications of the ACM, 7: 647–653; 1964.
16. Fu, K. Error-correcting parsing for syntactic pattern recognition. In Klinger, *et al.*, editors. Data structures, computer graphics and pattern recognition. New York: Academic Pres; 1976.
17. Glantz, H. On the recognition of information with a digital computer. Journal of the ACM, 4: 178–188; 1957.
18. Gorin, R. SPELL: spelling check and correction program. Referenced in [43].
19. Hall, P.; Dowling, G. Approximate string matching. Computing Surveys, 12: 381–402; 1980.
20. Horn, A. On sentences which are true of direct unions of algebras. Journal of Symbolic Logic, 16: 14–21; 1951.
21. Knuth, D. Sorting and searching. Reading, MA: Addison-Wesley; 1973.
22. Knuth, D.; Morris, J.; Pratt, V. Fast pattern matching in strings. SIAM Journal on Computing, 6: 323–350; 1977.
23. Levenshtein, V. Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl., 10: 707–710; 1966.
24. Litecky, C.; Davis, G. A study of errors, error-proneness, and error diagnosis in COBOL. Communications of the ACM, 19: 33–37; 1976.
25. Lowrance, R.; Wagner, R. An extension of the string-to-string correction problem. Journal of the ACM, 22: 177–183; 1975.
26. Morgan, H. Spelling correction in systems programs. Communications of the ACM, 13: 90–94; 1970.

27. Morris, R.; Cherry, L. Computer detection of typographical errors. IEEE Transactions on Professional Communication, PC-18, 1: 54–64; 1975.
28. Muth, F.; Tharp, A. Correcting human error in alphanumeric terminal input. Information Processing and Management, 13: 329–337; 1977.
29. Odell, M.; Russell, R. U.S. Patents 1,261,167 (1918) and 1,435,663 (1922).
30. Partridge, D.; James, E. Natural information processing. International Journal of Man-Machine Studies, 6: 205–235; 1974.
31. Peterson, J. Computer programs for detecting and correcting spelling errors. Communications of the ACM, 23: 676–687; 1980.
32. Peterson, J. A note on undetected typing errors. Communications of the ACM, 29: 633–637; 1986.
33. Pollock, J.; Zamora, A. Automatic spelling correction in scientific and scholarly text. Communications of the ACM, 27: 358–368; 1984.
34. Riseman, E.; Hanson, A. A contextual post-processing system for error-correction using binary *N*-grams. IEEE Transactions on Computing, C-23, 5: 480–493; 1974.
35. Robinson, J. A machine oriented logic based upon the resolution principle. Journal of the ACM, 12: 23–41; 1965.
36. Salton, G. Automatic information organization and retrieval. New York: McGraw-Hill; 1968.
37. Salton, G.; McGill, M. Introduction to modern information retrieval. New York: McGraw Hill; 1983.
38. Shaffer, L.; Hardwick, J. Typing performance as a function of text. Quarterly Journal of Experimental Psychology, 20: 360–369; 1968.
39. Siderov, A. Analysis of word similarity in spelling correction systems. Referenced in [2].
40. Szanser, A. Error-correcting methods in natural language processing. In: Information Processing 68. IFIPS; 1969: 1412–1416.
41. Thomas, R.; Kassler, M. Character recognition in context. Information and Control, 10: 43–64; 1967.
42. Tick, E.; Warren, D. Towards a pipelined Prolog processor. In: Proceedings of the 1984 international symposium on logic programming. Washington, DC: IEEE Computer Society Press; 1984: 29–40.
43. Turba, T. Checking for spelling and typographical errors in computer-based text. In: Proceedings of the ACM SIGPLAN/SIGOA symposium on text manipulation. New York: ACM; 1981: 51–60.
44. Ullman, J. A binary *N*-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. The Computer Journal, 20: 141–147; 1977.
45. Wagner, R.; Fischer, M. The string-to-string correction problem. Journal of the ACM, 21: 168–178; 1974.
46. Yannakoudakis, E.; Fawthrop, D. An intelligent spelling error corrector. Information Processing and Management, 19: 101–108; 1983.
47. Zamora, E.; Pollock, J.; Zamora, A. The use of trigram analysis for spelling error detection. Information Processing and Management, 17: 305–316; 1981.