Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 6. (Semi)automatic Name Correction

G. H. KIRBY,* M. R. LORD, and J. D. RAYNER

Department of Computer Science, University of Hull, Hull, HU6 7RX England

Received October 8, 1990

The problem is addressed of attempting to correct by computer software erroneous systematic organic chemical names. Software has been implemented and tested within the Hull University Nomenclature Translator so that the latter can handle submitted names found to be syntactically or semantically incorrect. A preprocessing stage before parsing removes many minor errors in punctuation. More substantial errors of syntax are detected during parsing and then processed. Errors in punctuation may be corrected by modifying the grammar rules to accept particular cases of erroneous syntax, which is then corrected by semantic code associated with the rules. However, this proved to be practicable for only a few constructions. The more general approach adopted involves a combination of spell-checking suspect fragments to find similarly spelled or sounding fragments, known to the translator, and referencing the grammar tables to determine whether any of these are acceptable in the context of the parsing to that point. Nonserious semantic faults are easily corrected, but faults with the magnitudes of locants and with their associated multiplying terms need interaction with the user. A successful correction rate of 70% was achieved on a small sample of externally generated incorrect names. The conclusion is that it is possible to semiautomatically correct, that is with some user interaction, many of the errors commonly made in systematic nomenclature. Errors that are not satisfactorily handled by the techniques adopted are discussed and suggestions made for improvement.

INTRODUCTION

Earlier papers in this series discussed the development of a formal grammar to describe some classes of IUPAC systematic organic chemical nomenclature[1] and the use of this grammar in software for syntax analysis (parsing) and semantic processing of chemical names.[2]

This paper addresses the problem of extending the software to assist users in correcting erroneous names. The problem arises from the work of the Chemical Nomenclature Advisory Service of the U.K. Laboratory of the Government Chemist (LGC). Systematic names are referred to LGC by industry, and by customs and excise, for verification or correction.

For some years, the Chemical Abstracts Service operated computer procedures to validate CA Index Names and names from the literature.[3] These procedures however were related to the CAS nomenclature, more tightly defined than IUPAC, and were based on ad hoc translation methods, introduced by CAS from 1967[4] and 1974.[5] The work now described was to find whether the formal grammar, and its associated dictionary of name fragments, could be used to correct the errors typically made in IUPAC systematic names.

Errors Made in Nomenclature. To assess the types of error most commonly made in systematic chemical nomenclature, 200 erroneous names of imported chemicals, found and subsequently corrected by the LGC, were obtained from their Special Chemical Return "A" and "B" Lists. By analyzing these names, the results in Table I were obtained.

Table I

type of error	names with this error: number	%
names incomplete/not specific	54	27
fragments clearly misspelled	18	9
names not in preferred form	26	13
syntax incorrect due to extra or missing commas, spaces, colons, hyphens and brackets	102	52
locants or multiplication terms missing or incorrect	89	45
total faults	289	
total names	200	
av faults per name	1.4	

These statistics clearly show the predomination of simple syntax errors concerned with punctuation, i.e., the nonalphabetic fragments, and semantic errors associated with locant number and multiplying term.

APPROACHES TO NAME CORRECTION

Many minor errors in punctuation, such as superfluous spaces and other symbols, are quite trivial to detect without the need for the parser to check the syntax against the grammar. A preprocessing stage before parsing can handle a number of these errors.

One approach to tackling more substantial syntax errors is to amend some of the grammar rules so that certain errors, such as punctuation errors, are nevertheless accepted at the syntax analysis stage but are processed by semantic code associated with the rules that permitted the syntax containing the errors. This approach has been investigated and is reported later. In practice, however, it proves useful only for a few constructions in the syntax of chemical nomenclature.

A more general approach is needed in which both punctuation and alphabetic fragments can be (semi-)automatically corrected within the syntax-analysis phase.

Spelling Correction. Computerized spelling-correction systems have been in existence for some time, and such systems are now used for a variety of different applications. Initially, spelling correction was focused mainly on output from optical character recognition (OCR) and voice recognition. Today, one of the largest application areas is word processing and desktop publishing (DTP).

Other applications that use such a facility include the following: name lists, e.g., names of passengers in airline reservation systems;[6] programming languages, e.g., spelling errors made within Interlisp-D programs;[7] and database text, e.g., SPEEDCOP (Spelling Error Detection/Correction Project), a Chemical Abstracts Service (CAS) project to provide spelling correction within scientific and scholarly databases.[8]

The power and reliability of a particular spelling correction program is directly related to the application for which it is used. This can be demonstrated by comparing a typical natural language spelling correction system for a word-processor with that used in a computer-programming language.

The former is certainly helpful in the preparation of a document, but it cannot be relied upon totally. While it can easily detect typographical errors, it cannot detect a word typed in the wrong place, a word accidentally left out, or a misspelled word which happens to be spelled in the same way as a completely different word.

In addition, the user can be presented with a large list of alternatives to a misspelled word, of which only one or two are actually valid at the point of error. Even worse is the case where a badly misspelled word produces a list of alternatives of which none are suitable, leading to potential confusion for the user. Thus, in many cases, an error taken in isolation cannot be fully considered without knowledge of its context.

Spelling-correction systems for programming languages are generally much more precise, as they are able to use knowledge of syntax to detect and correct errors. This has been demonstrated for the Interlisp-D programming language, in which there is a well-defined syntax and restricted vocabulary. Chemical nomenclature with our grammar and fragment vocabulary should therefore be suited to a spelling-correction system. Unlike programming and natural languages, however, name fragments (morphemes) will be spell-checked rather than whole words. The success achieved by this approach is described later.

CORRECTION OF SIMPLE LEXICAL ERRORS

Once a new name has been input, a preprocessing phase performs the following functions on the input name:

(1) Removal of all leading and trailing spaces.

(2) Replacement of all multiple, adjacent, and generally identical nonalphanumeric characters by a single character.

(3) Replacement of all noncompulsory capital letters by their corresponding lower-case letters.

(4) Checking for the correct number of brackets.

(5) Checking for the correct matching and type of bracket pairs, along with any necessary corrections.

All of the above tests can be carried out without the use of the SLR parsing tables.[2] Therefore, the parser (syntax analysis phase) does not waste time dealing with faults which can be corrected much more quickly and easily.

Any changes carried out automatically to the input name, that is, all but (4) above, are reported to the user. In order to correct a name with an odd number of brackets, (4), the parser must be involved for knowledge of the context. However, by the display of an appropriate error message, the user is warned of at least one syntax error before parsing commences. This message could well speed up the process of correction. If a bracket is one of the possible fragments allowed from the point of syntax failure, as listed by the syntax correction system to be described, there is a high likelihood that the absence of this bracket is the cause of the error.

GRAMMAR MODIFICATION

The first approach investigated for the handling of nontrivial faults involved modifying the grammar to allow certain commonly made punctuation errors to be accepted as valid by the syntax analysis phase. Then, during semantic processing, the semantic code for each of the modified rules is responsible for displaying a suitable error message and making the necessary correction to a faulty name. The semantic code for a rule is executed when the rule is reduced.

Figure 1 illustrates a simple grammar for hydrocarbons with halogenated derivatives, with rules for the nonterminals "sub-part" (substituent-part) and "loc-part" (locant-part) allowing for commonly made punctuation errors via the terminals "illegal-char", "illegal-char2", and "illegal-char3". For a detailed understanding of the whole of this grammar, and the unexpanded nonterminal "aliph-parent", the reader is referred to paper 2 of this series, and especially to Figure 4 therein.[1] Note that the syntax is constructed for analysis of names from right to left.

```
TERMINALS

aliph-root       =  meth ,  eth ,  prop ,  but ;
aliph-suffix     =  e ;
g-mult           =  mono ,  di ,  tri ,  tetra ;
gmult-mark       =  a ;
gmult-cont       =  pent ,  hex ,  hept ,  oct ,  non ;
val10to12        =  dec ,  undec ,  dodec ;
dec-mark         =  dec ;
val3to9          =  tri ,  tetra ,  penta ,  hexa ,  hepta ,
                    octa ,  nona ;
halogen-mark     =  fluoro ,  chloro ,  bromo ,  iodo ;
number           =  '0' ,  '1' ,  '2' ,  '3' ,  '4' ,
                    '5' ,  '6' ,  '7' ,  '8' ,  '9' ;
hyphen           =  '-' ;
comma            =  ',' ;
illegal-char     =  ' ' ,  ',' ,  '-' ,  '.' ;
illegal-char2    =  ' ' ,  '-' ,  '.' ;
illegal-char3    =  ' ' ,  ',' ,  '.' ;

RULES

name             =  aliph-name ;
aliph-name       =  aliph-suffix  aliph-parent  sub-part ;
aliph-parent     =  ...  ...  ...  ;
sub-part         =  illegal-part  simple-sub  sub-part ,
                    $ ;
illegal-part     =  illegal-char ,
                    $ ;
simple-sub       =  halogen-part  gm-part  loc-part ;
halogen-part     =  halogen-mark ;
gm-part          =  g-mult ,
                    gmult-mark  mult-value ,
                    $ ;
mult-value       =  gmult-cont ,
                    val10to12 ,
                    dec-mark  val3to9 ;
loc-part         =  hy-part2  locant  loc-seq  hy-part ,
                    $ ;
loc-seq          =  com-part  locant  loc-seq ,
                    $ ;
locant           =  number ;
com-part         =  comma ,
                    illegal-char2 ;
hy-part          =  hyphen ,
                    illegal-char3 ,
                    $ ;
hy-part2         =  hyphen ,
                    illegal-char3 ,
                    $ ;

ROOTSYMBOL          name
```

Figure 1. Simple grammar allowing incorrect punctuation.

In Figure 1, the rule with the lefthand-side (lhs) nonterminal symbol "illegal-part" allows any punctuation fragment to be present in a name between two adjacent substituents or between a substituent and the parent. This modification handles the case of extra space and/or hyphen fragments, often inserted when no locants are present, for example:

```
bromo difluoro-methane
     ^        ^
```

Both of the above errors would cause a reduction of the rule alternative "illegal-part = illegal-char", while a syntactically correct name would result in the rule alternative "illegal-part = $" (i.e., there is no illegal character present) being reduced. (Multiple illegal characters, e.g., a series of spaces, will already have been condensed to a single character by the preprocessor.)

The grammar allows for the presence of punctuation errors at both ends and within a locant clause. The rules with the lhs nonterminal symbols "hy-part" and "hy-part2" allow both the leftmost and rightmost ends of a locant clause within a name to contain a hyphen, an illegal character, or the null string. The terminal symbol representing the illegal character in these cases, namely, illegal-char3, does not contain the hyphen fragment.

Finally, the individual locant separator fragment can be represented by any punctuation symbol, rather than just a comma, through the "com-part" rule. In this case, the null string is not allowed as it would not be possible to determine if a locant comprising more than one digit is in fact two or more locants with missing separators.

The following similar names, which all have errors in punctuation, would be accepted by this grammar:

```
1.4,dichlorodec 2-ene
1.chloro,1.2,diiodotetradeca2,4dien-6yne
ethlyne
bromo difluoro.methane
```

The semantic code[2] for each rule or rule alternative allowing illegal punctuation comprises a procedure call with three parameters. These parameters include the position in the original input name where the error occurred, the necessary action to be taken (i.e., deletion, insertion, or replacement), and the correct punctuation fragment (for insertion and replacement only).

Thus, the error position in the original name can be indicated to the user with an appropriate message, while a copy of this name is corrected and a flag is set to indicate correction has taken place. When the parse path has been reexecuted, the original name is replaced by the copy if the flag has been set, and the user is shown the new name. If a name contains several punctuation errors, then they are each handled automatically in turn, starting with the rightmost one.

The above method of correction works well for punctuation fragments, as they are all only one character in length and they comprise a small subset of all the fragments in the grammar. However, a great deal of work is required in modifying a large grammar to allow errors to be accepted wherever punctuation is permitted. This is because, by allowing punctuation to be erroneous (extra, unwanted, or missing), the grammar tables produced by the SLR program[2] often contain parsing action table conflicts. It is usually possible to rewrite the rules to avoid these conflicts, using a skill developed from experience and practice. For example, some 30 parsing action conflicts are avoided by using "hy-part" and "hy-part2" in the "loc-part" rule of the grammar shown in Figure 1, rather than having two occurrences of "hy-part".

Other restrictions include the difficulty of expanding or modifying a grammar embedded with all the extra "correction" rules, especially if the modifications contain new punctuation, and the fact that the method cannot be extended to the large set of alphabetic fragments with several characters each. Also, any corrections that need to be made are performed in the semantic processing phase, rather than the syntax analysis phase as one would expect.

However the facility given by the above method, to trap certain extra or missing fragments such as the "a" and "e" vowel elision characters, is adopted in our correction software. As incorrect vowel elision is a semantic fault, the automatic trapping within the syntax analysis phase and correction within the semantic processing phase is most appropriate here.

SPELLING CORRECTION FOR NOMENCLATURE FRAGMENTS

The grammar-based nomenclature translator, with its relatively small fragment vocabulary, provides a novel environment in which to apply spelling-correction algorithms for misspelled alphabetic fragments in conjunction with the grammar rules for a complete syntax error correction system.

The technique which has been devised involves the precise identification of an erroneous fragment or character, obtaining near matches for that fragment which are grammatically acceptable up to that point in the name, then ranking the alternative corrections in order of decreasing likelihood, and offering the user either a list of alternative fragments to choose from or suggested names that are syntactically correct and are based on the ranking of corrections. The stages in this technique are described more fully in the remainder of this section and in further detail in the associated thesis.[9]

Precise Identification of a Suspect Fragment. One of the major differences between chemical nomenclature and other languages, such as natural and computer-programming languages, is the lack of delimiters between individual tokens. For example, compare fragments (morphemes) in chemical nomenclature with words in natural languages, separated by spaces or punctuation, and in computer-programming language text (reserved words, etc.), enclosed in quotes or separated by punctuation. In chemical nomenclature, the identification of individual fragments is much more complex, with names often containing strings of directly concatenated letters that comprise several functionally different fragments. However, the nonalphabetic fragments serve as delimiters where they are present.

This problem has been solved by character matching using two different data structures, depending upon whether a particular starting character is alphabetic or nonalphabetic. In the first case, successive letters in the name are matched against those in the tree-structured fragment dictionary until either there is no further path in the tree appropriate to the next input letter or there are no more letters in the sequence.[2] If any complete fragments have been identified at this stage, the longest one is used, by default, as the next input symbol to the parser. Backtracking allows for shorter fragments to be tried later if need be.

Nonalphabetic fragments are stored in a linear look-up table since, with the exception of locants, they are always only one character in length and contain no semantic information.

If a name is syntactically valid, a set of fragments with their corresponding terminal symbol codes and semantic information will eventually be produced, although a certain amount of backtracking may have been carried out. If a name contains a syntax error, then the backtracking process would eventually return to the rightmost end of the name, with no further alternatives being available (names are analyzed from right to left).

For a syntactically faulty name, a data structure "failure_store" holds information on the parse path in which failure occurred at the leftmost point into the name. There may be several such paths which fail at this same point, and information is stored on each.

By reference to any parse path in failure_store, two observations can be made. First, if the whole name had been processed before failure occurred, then it can be assumed that the name is incomplete and the error is one of omission, rather than the presence of a suspect fragment.

Secondly, the rightmost end of a suspect fragment can be identified by reference to the furthest fragment accepted before the point of failure. This suspect fragment could be misspelled, or there could be missing fragment(s) between it and the point of failure. In these cases, the complete suspect fragment needs to be isolated before a further investigation can be carried out.

As the location of the suspect fragment's rightmost character (RP, i.e., the Right Pointer) is already known, the problem involves determining the leftmost one (LP). Three conclusions can be drawn from the type of the known character pointed to by RP.

First, if the character is not alphanumeric, then it must be a punctuation character. As all punctuation is by single characters and erroneous multiple punctuation has been removed by the preprocessor, the complete fragment is now known, and LP is set to the same value as RP. Note that there may be other faults further on in the name, but these will not be dealt with until the current fault is corrected.

Secondly, if the character is a digit, then the suspect fragment is a number. As numbers can comprise more than one digit, a character search is made in the name until either a nonnumeric character is found or the complete name has been absorbed. LP is set to point to the leftmost digit found.

Alternatively, if the character is a letter, then the suspect fragment is alphabetic with an unknown length of possibly many letters. This case is potentially the most difficult. The solution adopted involves finding the next valid fragment in the name, if present, and assuming that it is legal in context. LP is then set to point to the letter at the right of its rightmost end.

The simplest case is where the next valid fragment is the null string, i.e., the whole name has now been absorbed. If so, the suspect fragment comprises a single letter, and LP is set to point to the same letter as RP. Otherwise, when there are unprocessed characters remaining in the name, the rightmost end of the next valid fragment is assumed at first to be one character to the left of that pointed to by RP, and a search is made in the fragment dictionary to find a valid leftmost end. Thus we assume initially that the suspect fragment is only one letter in length.

The searching process is trivial if the potential start character of the next valid fragment is nonalphabetic, but for an alphabetic potential start character, it and any successive letters are matched against those in the fragment dictionary.

If this search fails, the next valid fragment is assumed to begin two characters to the left of that pointed to by RP, and the searching is resumed. This process continues until either a valid fragment is found or the whole name has been absorbed without success. If no valid fragment can be found, then LP is set to point to the leftmost character in the name, so that the entire remainder of the name is taken as the suspect fragment.

Further processing is required if the next valid fragment comprises a single letter and there are still more unprocessed characters remaining in the name. Experience shows that such a single letter could nevertheless be part of the suspect fragment. This problem is solved by looking even further ahead in the name for another valid fragment, to the left of the recently found single letter fragment, and thus deducing whether the latter is a valid fragment or not.

The following example demonstrates the action taken when the fragment "stigmast" is misspelled in the name *2-chlorostigmastane*:

```
2-chlorostigmasstane
                ^RP       (initial state)
            LP^ ^RP       (intermediate state)
      LP^       ^RP       (final state)
```

In this case, the next possible valid fragment found to the left of the character pointed to by RP is "a". However, as no valid fragment is found to its left, the context indicates it not to be a complete fragment, and therefore searching is continued to the left. Eventually, "o" is accepted as the next valid fragment. The fragment to the left of "o" is "chlor", which is one of the acceptable fragments with "o" to its right. Thus, "o" is taken as the next valid fragment, and LP is set to point to the "s", as shown above, identifying the incorrect fragment as "stigmasst".

Once a suspect alphabetic fragment has been isolated from a name, additional procedures attempt to produce a set of fragments present in the current grammar and which closely resemble it, although they may not necessarily be valid in the current context. If any such matches are found, they are sorted into descending order of likelihood, before being further processed within the context-checking stage of the error correction system.

There are many different types of spelling errors that can occur in chemical name fragments. Two such types of error, namely, user ignorance and typographical errors, have been identified as being probably the most common among the 9% of names with misspelled fragments (Table I).

Obtaining Near Matches for a Suspect Fragment. Algorithms have been developed to produce a set of nearest match entries for a suspect fragment, using a "sounds familiar" and a typographical check, referred to as the "soundex check" and the "Damerau check", respectively. An associated spell-check dictionary is divided into two main parts, namely, "frag_spell_dict" and "soundex_spell_dict", which both contain a series of record entries. Each record in frag_spell_dict comprises a valid fragment and a list of all the SLR terminal symbols which contain that fragment. Each record in soundex_spell_dict comprises a four-letter soundex code and a list of all the valid fragments associated with that particular code.

Soundex Check. Errors due to user ignorance can lead to consistent misspellings, which are probably related to how a fragment sounds. For example, the fragment "naphthyl" could be misspelled as "napthyl", "naphthyle", "napthyle", "naphthile", "napthile", etc.

The soundex check tests to see if a suspect fragment "sounds similar" or is phonetically similar to any valid ones. The technique used is based upon the Soundex algorithm[10] and involves coding a suspect fragment into four letters, chiefly by dropping all vowels and replacing the consonants with one of six code letters, which each represent the sound of a group of consonants.

It is the length of the soundex codes used that helps to control the number and accuracy of the near matches produced for a particular suspect fragment. By trial-and-error investigation, it was discovered that four-letter soundex codes were most appropriate for nomenclature fragments.

Next, a similarity search is performed between the soundex code for the suspect fragment and the set of unique soundex codes for all the valid fragments in soundex_spell_dict. If the same code is found in soundex_spell_dict, the corresponding list of valid near-match fragments is obtained from the appropriate record. Finally, the set of terminal symbol classes is found, to which each of these valid fragments belong, by cross-reference to frag_spell_dict. For example, the soundex search code and the near matches obtained for the suspect fragment "napthel", found in the name *napthelene* (intended name: *naphthalene*), are shown below:

```
Soundex search code: "mptl".

Match (a) found: "naphthal"
   Terminal symbol(s): "trivarom-root" (naphthal, anthrac,
                                         phenanthr).

Match (b) found: "naphthyl"
   Terminal symbol(s): "triv-subD" (naphthyl, anthryl,
                                     phenanthryl).
```

It is match (a), "naphthal", that produces the intended fragment, found to belong to the terminal symbol class "trivarom-root" (trivial aromatic parent root), by cross-referencing the appropriate record in frag_spell_dict. The fact that the other match, (b), does not actually allow a meaningful name to be produced is not relevant at this stage, but is detected when context checking is performed.

Dameau Check. Typographical errors are not so consistent as those described in the previous section. However, they may well be more predictable, since they are related to the position of keys on the keyboard and often result from errors in finger movement. Damerau[11] investigated the types of spelling faults made by users of a text retrieval system. He discovered that 80% of those words rejected because of spelling errors were the result of either a transposition of two letters, one letter extra, one letter missing, or one letter wrong.

The Damerau check involves comparing minor variations of the suspect fragment against records in frag_spell_dict, containing a suitable length of valid fragment, on the assumption that one of the above four most common typographical errors has occurred. For example, the comparisons performed and any subsequent matches obtained for the suspect fragment "yr", found in the name *3-methy̲r̲heptane* (correct name: *3-methylheptane*), are shown below:

(1) *Assume two letters have been transposed*
Compare "ry" with all valid fragments in frag_spell_dict with a length of 2 letters.
No matches found.

(2) *Assume one letter is extra*
Compare "y" and "r", in turn, with all valid 1-letter fragments.
No matches found.

(3) *Assume one letter is missing*
Compare ".yr", "y.r" and "yr.", in turn, with all valid 3-letter fragments, where "." means "match any single letter in this position".
Match (*a*) *found*: ".yr" (where "." matches "p").
Terminal symbol(*s*): "arom-root" (triphenyl, chrys, naphthac, *pyr*, etc.).

(4) *Assume one letter is wrong*
Compare ".r" and "y." with all valid 2-letter fragments.
Match (*b*) *found*: "y." (where "." matches "l").
Terminal symbol(*s*): "rad-mark" "conj-radmark" (y̲l̲).
Match (*c*) *found*: "y." (where "." matches "n")
Terminal symbol(*s*): "yn-mark" (y̲n̲).

It is match (b), "yl", that produces the intended fragment. This fragment is also found to belong to two separate terminal symbol classes, by reference to the appropriate record in frag_spell_dict, of which "rad-mark" (the radical suffix) is the appropriate symbol here. None of the other matches produce a meaningful name when context is checked.

If the wrong letter assumption, (4) above, produces a near match, a further test, known as the *keyboard adjacency test*, is carried out to see if the suspect letter is adjacent on the QWERTY keyboard to the corresponding one in the near-match fragment. If such a test proves to be true, the match is assigned a higher priority.

For example, from checking the suspect "yk" fragment, found in the name *3-methy̲k̲heptane*, one of the near-match entries will be the fragment "yl" (i.e., match found with "y.", as shown in the previous example). In addition, the keyboard adjacency test will be successful, as the letter "k" (in the suspect fragment "yk") is located on the keyboard immediately to the left of the letter "l" (in the valid fragment "yl").

The above test could be expanded to detect other types of wrong letter error, such as the reversal stroking error made by typists. This error involves accidental substitution of a left-hand stroke for a right-hand stroke, or vice versa. For example, the letter "e" could be typed instead of the letter "i", by using the middle finger of the left hand instead of the middle finger on the right hand.

Table II

name	deccane	deecane
suspect fragment	c	deec
near match fragments	R S a b d etc.	dec dioic etc.

Performance of Spell-Check Software. In order to test the average number of near matches produced for a typical suspect fragment, containing one of the four common typographical errors, a randomly generated set of 250 erroneous fragments was used as input to the Damerau check software. From this sample, 203 erroneous fragments (81.2%) resulted in one match, 33 erroneous fragments (13.2%) resulted in a choice of two matches, and in only 14 cases (5.6%) were more than two matches obtained.

The maximum number of comparisons required, using the Damerau check, for a suspect fragment of length N with the valid fragments in frag_spell_dict is $a + b + c + d$, where
$a = (N - 1) \times$ number of fragments with length N
(the number of comparisons for two letters transposed);
$b = N \times$ number of fragments with length $(N - 1)$
(the number of comparisons for one letter extra);
$c = (N + 1) \times$ number of fragments with length $(N + 1)$
(the number of comparisons for one letter missing);
$d = N \times$ number of fragments with length N
(the number of comparisons for one letter wrong).

Although this can result in a large number of possible comparisons, between 650–1000 for fragments of commonest length 4–8 characters, it is manageable. For example, an Opus PCII (IBM PC XT compatible) running at 8.0 MHz took 147 s to check all the 250 erroneous fragments, described above, using a combination of all the checks. This gives an average time of 0.6 s to check a single erroneous fragment, including input of the fragment and output of the results. Although this time period is negligible, it has to be remembered that such an operation is only a small part of the overall correction process, and thus, it raises questions about the viability of this approach for handling two or more mistakes in one fragment.

Each unique near-match fragment found is assigned a probability according to its similarity to the suspect fragment. The highest probability applies when only one letter is different, with the keyboard adjacency test true, and the soundex codes the same. The lowest probability occurs when only the soundex codes match, with the fragments differing in length by more than one letter.

If the suspect fragment is actually valid and known, an appropriate flag is set. This allows the user to be informed, within the context checking stage, that the suspect fragment has either been accidentally spelled in the same way as a correct one or is currently in an illegal position in the name.

The combined systems that isolate a potential suspect alphabetic fragment and generate a set of near-match fragments work well. It is difficult to collect large numbers of test names with misspelled fragments, which represent typical mistakes made by users and which belong to those classes of compound that can actually be translated by the still developing nomenclature translation system. A sample set of approximately 50 names containing such faults resulted in a successful correction rate of almost 70%. These names were obtained from the LGC's Special Chemical Return "A" and "B" Lists, together with a file of names produced after 20 chemistry students, from a local school, had been given access to the system.

Obviously, there are complications if, say, a suspect fragment has not been isolated correctly. The location of an error in a fragment is more important than it might at first seem. For example, consider the two similar erroneous names in Table II, each containing one extra letter.

Here, both names result in syntax failure after the saturation term, "an", has been absorbed. However, the next valid fragment for *deccane* is "dec" and for *deecane* the null string. Thus, they each produce different suspect fragments, "c" and "deec", which result in different near-match fragments being produced, as shown in Table II. In fact, both these names can be corrected from this information within the context-checking stage (see later), although in the first case the near-match fragments are of no value.

To improve on this, an additional feature has been implemented whereby a short suspect alphabetic fragment is combined in turn with the alphabetic fragment to its left and right as appropriate, with the conjoint fragment used as input to the near-match algorithms, i.e., 'decc' and 'can' for *deccane*. Where present, the indexes of a left and right alphabetic fragment are determined from the search for the suspect fragment's left pointer (LP), and the parse paths stored in failure_store, respectively. This facility alleviates the problem of unreliable near-match fragments being produced, and therefore provides greater consistency.

CONTEXT CHECKING

Normally, an SLR parser is deterministic, that is, it does not backtrack. However, the parser used to recognize the chemical nomenclature grammar does need to backtrack, to handle situations where, for example, the wrong fragment is extracted as the next input symbol (by default, the longest fragment is tried first) or where the wrong terminal symbol interpretation is chosen for the fragment (since a fragment may belong to more than one acceptable terminal symbol class).

A syntactically erroneous name leads eventually to failure of all the backtracking alternatives. For such names, the failure_store data structure allows first the identification of the suspect fragment for what may be more than one reliable interpretation of the name thus far. Secondly, the context(s) into which any replacement fragment must fit is(are) known.

For each of the reliable interpretations, the set of terminal symbols allowable from the point of failure is obtained from the parse tables and is checked for validity within the context of the current fault. Next, a check is made to determine whether any of the near-match fragments found in the spelling-check process, already described, belong to any of the valid terminal symbols. Finally, the whole set of valid terminal symbols is obtained.

SYNTAX CORRECTION

A syntax error correction interface has been added to the nomenclature translator, which offers the user either the information from which to make a semiautomatic correction or the choice of automatic correction mode.

In the former case, the erroneous fragment is clearly identified within the input name. For each reliable interpretation, the terminal symbols allowed at the point of failure are displayed along with their associated fragments. Among these, any near-match fragments from the spelling check are highlighted. Additionally, messages are displayed as appropriate, to inform the user that

(1) there may be a missing fragment—if the suspect fragment is a valid fragment
(2) the suspect fragment may be superfluous—especially if it is only one character in length
(3) there is no suspect fragment—if the name is incomplete
(4) no near matches found—if the suspect fragment does not closely resemble any of the fragments in the set of valid ones.
(5) only alphabetic fragments can be spell-checked—if the suspect fragment is nonalphabetic

```
IF ( initial ) suspect fragment has a length of 1
THEN ( It may be superfluous )
   check_syntax(deletion) ( Check the syntax after
             the suspect fragment has been deleted )
ENDIF;


FOR each appropriate suspect fragment DO
   <spell- and context-check initialisations>

   FOR each valid interpretation DO
      IF near match terminal symbol(s) found
      THEN ( Assume error due to a faulty/missing
                near match alphabetic fragment )
         FOR each near match fragment DO
            check_syntax(replacement)
         ENDFOR

      ELSE ( Assume error due to a punctuation fault )
         FOR each punctuation symbol allowed DO ( From
                                 the point of failure )
            IF no suspect fragment ( Incomplete name )
               OR suspect fragment is valid
            THEN
               check_syntax(insertion)
            ENDIF;
            IF suspect fragment found
            THEN
               check_syntax(replacement)
            ENDIF
         ENDFOR
      ENDIF;

      IF the eoi terminal symbol allowed
      THEN ( The name may already be complete here )
         check_syntax(eoi)
      ENDIF
   ENDFOR;

ENDFOR;
```

Figure 2. Algorithm for automatic name correction.

Insertion, deletion, and replacement operations are available to the user. Once the name has been modified, the new version replaces the original input name and a full reparse is initiated. If another syntax error is found (further to the left than before), the error correction system will be invoked again.

In automatic correction mode, the system attempts to produce a set of syntactically correct names from the faulty one. Each of these names can then be viewed in turn, and if desired, the one currently being displayed can be selected for parsing in place of the original input. For each corrected name, the user is informed as to how the original faulty name has been modified. The pseudo-Pascal algorithm shown in Figure 2 illustrates the order in which checks are applied in the search for correct names.

The procedure "check_syntax" modifies a copy of the original name according to the type of action specified by the parameter. Then, a parse of the complete name is carried out to determine if it is syntactically correct, in which case the name is added to the set. Note that the parse is not necessary with the special end-of-input case, as the modified name here is guaranteed to be syntactically correct.

At present, only one error is handled by this method, although extending it to handle two or more errors could easily be implemented. Obviously, the time taken and the number of corrected names found are directly related to the number of errors that the system attempts to correct. Although it is possible for the user, through interaction with the system, to semiautomatically correct names with many faults, the automatic system must be more limited.

It can be argued that if a name has three, or even two, syntactic errors, there is too much wrong for an automatic correction system to attempt to put right. Then the user must be left to make corrections, albeit with as much help as the system can provide.

The assumption that a fault can be corrected using only the near-match fragments, if any are found, is important. Not only do these fragments have a higher probability of producing the user's intended name than the others, they also permit a corrected name to be presented to the user without the need for any semantic checks. If all the non-near-match fragments of terminal symbols allowed from the point of failure were considered, then one would expect most of these to produce names that were not what the user had originally intended, and many more of them.

Punctuation errors are more straightforward to handle, because there is only one fragment per terminal symbol. Also, as these fragments are only one character in length, a name corrected with such symbols has a high probability of being the user's intended one.

Interaction with the user is necessary if a particular correction requires the addition of a locant. For such cases the modified name presented to the user contains the dummy locant string. If this name is subsequently selected, the user must enter a suitable locant value to replace the dummy string.

If the faulty name *4-(1,2-difluoropropyl)didec-2-ene* is submitted to the parser, "dec-2-ene" will be accepted and the fragment "di" will be identified as the initial problematic fragment. Only one correct name, namely, *dec-2-ene*, is produced by invoking automatic correction mode in the current context. This is obtained by deleting all the fragments to the left of the point of failure, as a result of the special end-of-input (eoi) terminal symbol being allowed here. By using a combination of the "di" and "dec" fragments, another valid name is produced by the automatic mode software, namely, *4-(1,2-difluoropropyl)dodec-2-ene*. This more realistic correction results from a replacement operation of the new suspect fragment ("didec") with the near match and valid in context "dodec" fragment.

The apparently similar faulty names *deccane* and *deecane*, already described, both result in the one correct name, *decane*, being produced via the automatic correction mode. In the latter case, the method of correction involves replacing the suspect fragment ("deec") with the near match and valid in context "dec" fragment. However with the former, one solution is found by deleting the superfluous initial suspect fragment, "c", and another solution results from replacing the combination suspect fragment, "decc", with the near match and valid in context "dec" fragment.

SEMANTIC CORRECTION

A semantic error correction system that automatically corrects nonserious faults and corrects certain serious faults, via interaction with the user, has been implemented and tested.

Semantic faults that can be corrected automatically include incorrect use of the vowel elision fragments "a" and "e", locants in nonascending numeric order and substituents in nonalphabetic order.

The problem of incorrect vowel elision is handled by having extra grammar rules that allow the "a" or "e" fragments to be present or not within the appropriate clauses. Any such faults are then automatically corrected during the reexecution of the parse path in the semantic-processing phase, using the method described earlier in this paper. For example, the name *icos-2,4-dien-7-yne-11-ol* will automatically be corrected to *icosa-2,4-dien-7-yn-11-ol*, accompanied by the production of suitable messages to inform the user that the "e" should not be present to the right of "yn" and that there should be an "a" to the right of "icos".

During the reexecution of the parse path,[2] a temporary linked list of locant values is constructed each time a locant clause is encountered. This is then used in the processing of the semantic and substituent trees. Thus, a check to find out if the values are in the correct numeric order is made using this temporary data structure, and any errors found are subsequently corrected by performing a sort operation.

Further use of the above data structure allows a suitable message to be output if no locant is present for a single structure, such as a substituent, informing the user that a default value of one is assumed.

Checking and correcting the alphabetic ordering of substituents is carried out when the entire name has been processed by reference to the substituent tree.

A frequent semantic error that cannot be corrected automatically is that of a mismatch between a multiplier term and its number of locants. Such errors are reported to the user and are corrected through system/user interaction. For example, if the user confirms the locants as originally given, the system can correct the multiplier automatically. Incorrect locant values similarly require user assistance for adjustment after the system has identified the faulty locant(s) and the parent structure.

DISCUSSION AND CONCLUSION

The evidence such as we have been able to gather points to syntax errors, predominantly in punctuation, being the most common fault in chemical names. The other major cause of errors is the locant and multiplier terms being incorrect or missing.

It is clear from the work reported here that the use of additional syntax rules to help in correcting the syntax of erroneous names is valuable for correction of punctuation, but not for more general use given the number of grammar rules needed to describe some classes of nomenclature. The application of the spell-checking ideas from word processing to the fragments of a chemical name, coupled with context checking using the grammar rules, can however handle syntax errors resulting from faults with all types of fragment. This also involves no changes to the grammar, and all corrections are performed within the syntax-analysis phase.

The latter approach is not guaranteed to correct certain simple typographical errors in all cases. For instance, if the transposition of two letters occurs across the boundary of two fragments, e.g., *deacne* for *dec|an|e*, the suspect fragment, in this case "deacn", is unlikely both to be a near match to the required fragment ("an" here) and to match those fragments allowable from the point of failure. The semiautomatic correction system, though, will indicate the allowed terminal classes from the point of failure, from which an intelligent user might realize the error. However, the three other common types of typographical errors tested for in the Damerau check would be corrected in most cases at fragment boundaries.

Automatic means of coping with the transposition error across fragment boundaries can be conceived. For instance, if spell and context checking fail to give a replacement fragment and the suspect fragment is four or more letters long, the splitting of this in all possible ways into two fragments of length two or more letters, followed by spell and context checking of these from the right, could be attempted.

The design of the grammar has a marked effect on the behavior of the syntax correction system. Factors that influence the design include the construction of the rules (i.e., top-down, bottom-up, or a combination) and the fragment set. The grammar used[1] was not designed with error correction in mind, and, with hindsight, modifications can be seen which would assist the error-correction process. For example, in the fragment set, should the 12-atom multiplier term be represented by the single fragment "dodec" or by two separate fragments "do" and "dec"? The additional facility of combining a suspect fragment with one of its neighbors and treating the combination as another possible alphabetic suspect fragment has proved useful with this problem.

Potential confusion can arise when different terminal symbols allowed from the point of failure contain similar fragments. For example, the terminal symbols "aliph-suffix" and "steroid-suffix" both contain the fragment "e"; "ol-mark" (the alcohol suffix) and "conjol-mark" (for conjugated alcohols) both contain "ol"; while the fragment "dec" appears in both "val10to12" (multiplier values for 10–12 atoms) and "decmark" (leftmost fragment for multipliers from 13 to 19).

The number of terminal symbols that is allowed from a point of failure and, hence, the numbers of reliable interpretations and possible correct fragments are dependent on where in a name the failure occurred. For example, if failure occurred within a multiplying term clause, there would be few symbols allowed. If however the failure occurred at a point where a substituent clause could be present, then there would be many terminal symbols to choose from.

In the semantic-correction process, there are several more sophisticated investigations that could be added, at the expense of more complex code and increased processing time. For example, in the case of invalid locants, consider the name *12-dichloropentane*. Here, the following deductions could be made concerning the faulty halogen substituent:

(1) The locant value of 12 is too large for the parent aliphatic chain length of five carbon atoms.

(2) The explicit multiplier "di" implies that the user probably intended to give more than one locant. (From our experience, it is quite common for multiplier terms to be omitted, but the presence of unnecessary ones is rare.)

(3) A valid name is created if the 2-digit number "12" is converted by insertion of a comma to create two locants, i.e., "1,2-...".

Thus, the user could be shown the valid name *1,2-dichloropentane* and asked if that is what was intended.

With this more computationally intensive type of correction, it is important for the system to give up if no solution has been found after some predetermined length of time, or when a certain level of detail has been reached. Nevertheless, as processing power of PCs increases, the number and complexity of attempts to correct erroneous names can expand.

It will be a definite advantage to automatic correction, if a semantic check is applied where two or more alternative syntactically correct names have been constructed by the syntax error correction system. For example, consider the name *pentane-1,3-dol*, in which the following two syntactically correct names are automatically produced by the system described:

```
pentane-1,3-diol     (Replacement of "d" by "di")
pentane-1,3-ol       (Deletion of "d")
```

By applying a semantic check on each of these names, it can be seen that while the first is acceptable, the second has a locant/multiplier term conflict in the position where the syntax correction has just taken place. Thus, both names can be presented to the user, along with a suitable message explaining why the system believes the first to be the more likely candidate.

The principal conclusion of this work is that it is possible to correct semiautomatically, i.e., with some limited input from the user, many of the errors commonly made in the use of systematic chemical nomenclature. An automatic correction mode can provide one or more names that are syntactically correct, but some user judgment is necessary to determine whether these include the intended name.

Applications of the software we have developed could include industrial/commercial data entry and validation of names where input is by nonchemically trained personnel and for computer-based teaching of chemical nomenclature.

ACKNOWLEDGMENT

We are pleased to acknowledge the support provided by the U.K. Laboratory of the Government Chemist and the helpful comments of Mr. I. Cohen of its Chemical Nomenclature Advisory Service on this work.

REFERENCES AND NOTES

(1) Cooke-Fox, D. I.; Kirby, G. H.; Rayner, J. D. Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 2. Development of a Formal Grammar. *J. Chem. Inf. Comput. Sci.* **1989**, *29*, 106-112.
(2) Cooke-Fox, D. I.; Kirby, G. H.; Rayner, J. D. Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 3. Syntax Analysis and Semantic Processing. *J. Chem. Inf. Comput. Sci.* **1989**, *29*, 112-118.
(3) Vander Stouw, G. G. Computer Programs for Editing and Validation of Chemical Names. *J. Chem. Inf. Comput. Sci.* **1975**, *15*, 232-236.
(4) Vander Stouw, G. G.; Naznitsky, I.; Rush, J. E. Procedures for Converting Systematic Names of Organic Compounds into Atom-Bond Connection Tables. *J. Chem. Doc.* **1967**, *7*, 165-169.
(5) Vander Stouw, G. G.; Elliott, P. M.; Isenberg, A. C. Automated Conversion of Chemical Substance Names to Atom-Bond Connection Tables. *J. Chem. Doc.* **1974**, *14*, 185-193.
(6) Davidson, L. Retrieval of Misspelled Names in an Airlines Passenger Record System. *Commun. ACM* **1962**, *5*, 169-171.
(7) Xerox Corporation. *Interlisp-D Reference Manual*, Volume II: Environment. Xerox Corp.: 1985; Chapter 20.
(8) Pollock, J. J.; Zamora, A. Automatic Spelling Correction in Scientific and Scholarly Text. *Commun. ACM* **1984**, *27*, 358-368.
(9) Lord, M. R. Name Correction and User Interface Enhancements in a Chemical Nomenclature Translator. PhD Thesis, University of Hull, Hull, England, 1990.
(10) Munnecke, T. Give your Computer an Ear for Names. *Byte* **1980**, *May*, 196-200.
(11) Damerau, F. J. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM* **1964**, *7*, 171-176.

DETHERM: Thermophysical Property Data for the Optimization of Heat-Transfer Equipment

DAVID F. ILTEN

DECHEMA, Theodor-Heuss Allee 25, D6000 Frankfurt am Main 97, Germany

Received October 11, 1990

A physical-property calculation system coupled with a very comprehensive and extensive literature database is described. Methods and methodologies for generating the thermophysical property data necessary for carrying out detailed and accurate design and check-rating calculations of heat-transfer equipment are discussed.

INTRODUCTION

With the ever-increasing sensitivity of economic and pollution issues, the requirements for the performance of heat-generation and heat-transfer equipment are becoming continually more stringent. The "overall" methodologies and algorithms with LMTD values and correction factors traditionally used for the mathematical modeling of fired heaters,