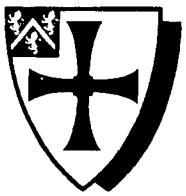
Spelling correction in the NLP system 'LOLITA: dictionary organisation and search algorithms

Brett Stephen Parker



University of Durham

Spelling Correction in the NLP System 'LOLITA': Dictionary Organisation and Search Algorithms

Brett Stephen Parker

Laboratory for Natural Language Engineering,
Department of Computer Science,
University of Durham, U.K.

Submitted for the degree of Master of Science, September 1994

Abstract

This thesis describes the design and implementation of a spelling correction system and associated dictionaries, for the Natural Language Processing System 'LOLITA'. The dictionary storage is based upon a trie (M-ary tree) data-structure. The design of the dictionary is described, and the way in which the data-structure is implemented is also discussed. The spelling correction system makes use of the trie structure in order to limit repetition and 'garden path' searching. The spelling correction algorithms used are a variation on the 'reverse minimum edit-distance' technique. These algorithms have been modified in order to place more emphasis on generation in order of likelihood. The system will correct up to two simple errors (*i.e.* insertion, omission, substitution or transposition of characters) per word. The individual algorithms are presented in turn and their combination into a unified strategy to correct misspellings is demonstrated. The system was implemented in the programming language Haskell; a pure functional, class-based language, with non-strict semantics and polymorphic type-checking. The use of several features of this language, in particular lazy evaluation, and their corresponding advantages over more traditional languages are described. The dictionaries and spelling correcting facilities are in use in the LOLITA system. Issues pertaining to 'real word' error correction, arising from the system's use in an NLP context, are also discussed.

Acknowledgements

Firstly I would like to thank my supervisor, Roberto, for his help throughout the year, and for getting me here in the first place. Also the other members of staff in the LNLE; Rick, Russell, Deborah and Chris, for all of their help and assistance.
I would like to thank everyone in the 'AI lab' (Dave, Jon, Sengan, Simon, Stephen and Yang) for their help in proof reading and bringing me up to speed with LOLITA and Haskell, as well as the rest of the group (Agnieszka, Kevin, Mark, Miguel and the two Pauls).
Karen Kukich at Bellcore and Roger Mitton at Birkbeck College for their advice and help in locating some difficult to find information.
Special thanks to Gillian for her many paper-hunting visits to the JRULM and for her continuous support. Finally, I would like to thank my family for all of their encouragement, without which none of this would have been possible.

Contents

List of Figures

Chapter 1

Introduction

In this thesis a method of dictionary storage and access will be introduced, and its use in solving the problem of correcting spelling mistakes in text described. This chapter introduces the problem of spelling correction and examines, briefly, some of the issues involved in dictionary organisation.

1.1 Problem Outline

Humans are able to deal with incorrect spellings when reading text. It is perfectly possible for a person to read a passage of text containing spelling errors, never realising that as they do so they are continually making mental corrections. It is this ability to correct 'on the fly' that a machine must try to mimic if it is to perform at a level approaching human capabilities at text processing. It will be shown, throughout the course of this work, that the spelling correction problem is not as simple as human behaviour would suggest. Indeed Chapanis [CHA 54] states that "random mutilation by deleting letters can not exceed 25% before humans fail to restore text" (as cited by [HAR 72]). Central to the development of a spelling correction system is the dictionary of which it makes use. The dictionary will be used to check the validity of words in the input stream and to test any words

hypothesised by the spelling correction system. Clearly, the performance of the spelling corrector is closely allied to that of the dictionary, so that any assistance which one can give the other will enhance the overall execution speed.

1.2 Spelling Errors

Before a method for correcting errors can be developed it is first necessary to examine the source of the errors as they are observed to occur. Damerau, in his seminal 1964 paper [DAM 64], put forward a classification of errors which has become the de-facto standard for describing simple errors. He claimed that 80% of errors could be classed into the following four groups:

1. Insertion – An extra character is inserted into the string, e.g. the word 'spell' becomes 'speell' with the insertion of an extra 'e'.

2. Omission – A character has been missed out of the word, e.g. the word 'spell' becomes 'spel' with the omission of an 'l'.

3. Transposition – Two adjacent characters have been swapped, e.g. the word 'spell' becomes 'sepll' with the transposition of the second and third characters.

4. Substitution – A character has been replaced with another, e.g. the word 'spell' becomes 'srell' if an 'r' is substituted for the 'p'.

Peterson [PET 80] gives a diagrammatic representation of the way in which a word can be transformed by these errors, which is reproduced in Figure 1.1.

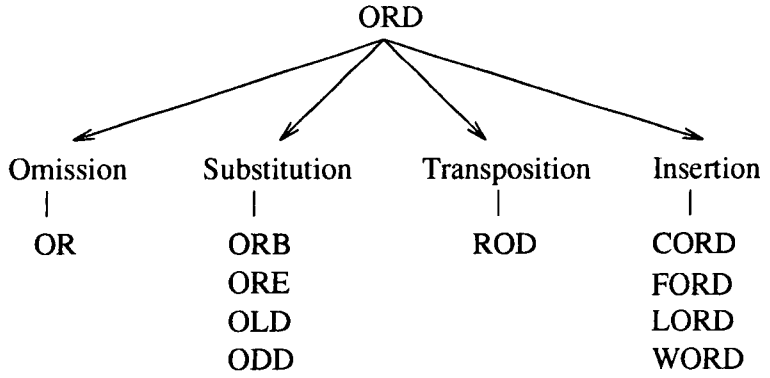


Figure 1.1: Peterson's example of error transformations.

It has been shown how a word could be corrupted by each of these errors, but why are the errors made in the first place? Analysis of the problem will be divided into two parts; convention errors and slips.

1.2.1 Convention Errors

A 'convention error' will be defined as one in which the writer does not know the spelling of a word. There is not, in English, a direct mapping between phonemes (sounds) and letters, the former outnumbering the latter almost two to one. There is, therefore, a major difficulty if attempting to spell by ear as can be seen in the following example [CRY 87]: *sheep* has only one possible pronunciation, 'Sip' whereas the phoneme 'Sip' has three possible spellings, sheep, sheap and shepe. The types of errors which arise as a result of phonetic confusion are examined further in Section 5.2.3.

1.2.2 Slips

'Slips' [HOT 80] are defined as errors in which the writer does know the correct spelling of a word, but fails to type correctly.

It is assumed, in this work, that the primary method of data entry, the man-

machine interface, is a standard QWERTY keyboard, as shown in Figure 1.2.

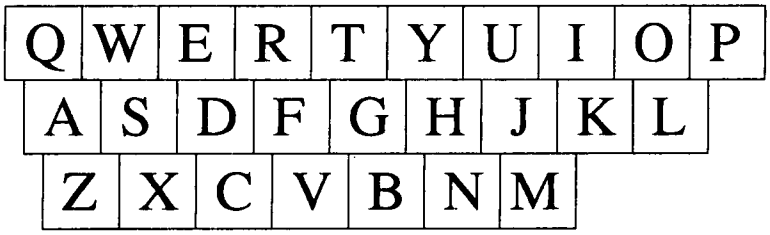


Figure 1.2: The standard QWERTY keyboard

The method of entry has an important impact on the types of errors which are found. A high number of substitution errors would be expected on a piece of text which has been scanned using an Optical Character Recognition system, for instance. Whilst the implementation of the system which has been developed is influenced by this factor, it will be shown later that the adaptation of a different input method would require a relatively trivial adjustment to the penalty weighting associated with the various error forms. There exists a direct correspondence between Damerau's classifications and the mechanics of keyboard entry,

1. Insertion – Two keys are struck simultaneously when intending to hit one.
2. Omission – A key is not depressed firmly enough.
3. Transposition – Keys are hit in the wrong order.
4. Substitution – The wrong key is struck.

These observations suggest spelling error patterns which will be exploited by the algorithms for spelling correction presented later. Section 5.2 provides a more detailed analysis of typing errors.

1.3 Dictionaries

The issue of dictionary organisation is of fundamental importance in the design of a spelling correction system. The dictionary can vary in complexity from being

simply a word-list, to some of the more advanced schemes described in Section 3.3.

Briefly, the main considerations during the design of dictionaries are: what information is to be kept; how this information is to be accessed; and at what costs. The type of dictionary which is used in everyday life, a paper book, is actually quite a complex structure to emulate. Access is quasi-direct, it contains tens, or often hundreds, of thousands of words, and each word entry contains details on meaning, pronunciation and usage. The dictionary used in a spelling correction system for a simple word-processor would not need any information except whether a word exists or not, a speech synthesiser may require only the pronunciation, whereas a natural language processor will need the meaning.

The way in which a dictionary is organised will be influenced, also, by the demands placed upon it by the rest of the system. A trade-off may be made between speed of access and the space in memory which the dictionary uses, for example. A system which operates in a batch environment will have different demands to a system which works interactively, in which a pseudo-real time performance will be required. Clearly, the development of a dictionary system first needs a thorough analysis of the requirements, both current and forecasted, in order to avoid the system becoming obsolete because it cannot cope with its environment. The issues affecting the design of a dictionary are discussed in more depth in Chapter 4.

1.4 Summary

In this chapter the ideas upon which the work presented in this thesis builds have been introduced. The problem to be addressed by this work was identified, namely how to produce a spelling correction system and associated dictionary storage. The two main aspects of the work, spelling error correction and dictionary organisation, were discussed in outline. The spelling corrector and dictionary will both be looked at in greater detail in the following chapters.

Chapter 2

Context of this work

The work presented has been influenced by the environment in which it has been developed. The Natural Language Processing system LOLITA, for which the spelling correction system has been developed, is described in order to show the practical applications of the work. The use of Functional Programming, in the implementation of the solution, has enabled the use of several important features not found in most programming languages. Finally the methodology of Natural Language Engineering, which has been used throughout the development of LOLITA, is presented.

2.1 The LOLITA NLP System

The LOLITA (Large-scale Object-based Linguistic Interactor Translator and Analyser) system [GAR 92] is a state of the art natural language processing system, able to grammatically parse, semantically and pragmatically analyse, reason about (see [LON 94]) and answer queries on normal complex texts, such as articles from the financial pages of quality newspapers. Begun in 1986, the system is being developed by the Laboratory for Natural Language Engineering at the University of Durham, currently involving a team of approximately twenty developers. In June

1993 the LOLITA system was demonstrated to the Royal Society in London.

The overall structure of the LOLITA system can be seen in Figure 2.1.

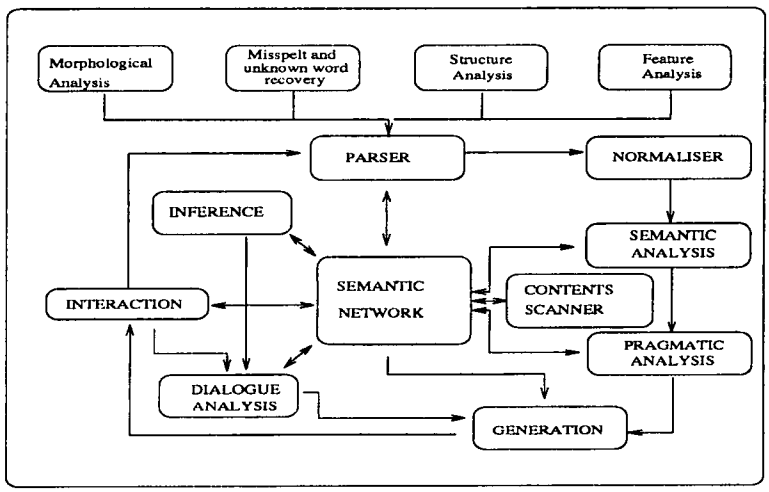


Figure 2.1: Structure of the LOLITA system.

The core of the system is a general framework which is used to map from text to meaning and meaning to text. The main data structure used to represent this meaning is a *semantic network* [SHA 88]. This structure holds world information and data, as well as some linguistic information.

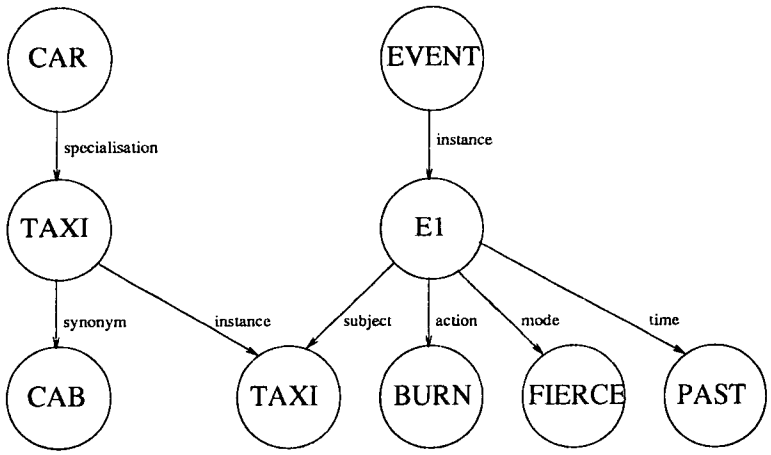


Figure 2.2: A portion of the semantic network

Figure 2.2 shows a simplified portion of the semantic network representing the event 'the taxi burned fiercely'. The transformation from text to meaning is carried

out by the parser, the normaliser, the semantic analyser and the pragmatic analyser. Each language understood by LOLITA (currently English, with some Italian and Chinese) requires the construction of a syntactic parser [ELL 93] to map from text to semantic net. The English parser, for example, contains over 1,500 grammatical rules. After syntactic parsing, the parse tree is normalised—certain parse trees which are equivalent are mapped to a unique normal form. The semantic analyser then transforms the parse tree provided by the normaliser into a fragment of semantic net, and matches the nodes it creates with those which currently exist. The pragmatic analyser then ensures that the meaning produced by the semantics is consistent with LOLITA's knowledge of the real world.

2.1.1 Applications of LOLITA

The core LOLITA system has been adapted for use in many different applications. These include contents scanning and Chinese tutoring described below, as well as dialogue analysis [JON 93] and generation [SMI 94], query application and machine translation.

Contents scanning

Contents scanning involves a passage of text being examined and information contained in the text being used to fill in an outline template. Contents scanning [GAR 93] is one of the standard tests of the abilities of a natural language processing system. The most widely known and acknowledged of such tests is that of the Message Understanding Conference (MUC), run by DARPA in the United States[1] [DAR 91]. An example of contents scanning, as performed by LOLITA, is given in Figure 2.3.

[1]LOLITA has been entered into the MUC-VI competition.

A car bomb exploded outside the Cabinet Office in Whitehall last night, 100 yards from 10 Downing Street. Nobody was injured in the explosion which happened just after 9 am on the corner of Downing Street and Whitehall. Police evacuated the area. First reports suggested that the bomb went off in a black taxi after the driver had been forced to drive to Whitehall. The taxi was later reported to be burning fiercely.

(THE DAILY TELEGRAPH 31/10/92)

```
Template: Incident
Incident: A bomb explosion.
Where   : On the corner of Downing Street and Whitehall.
          Outside Cabinet Office and outside 10 Downing Street.
          In a black taxi.
When    : 9pm.
          Past.
          Night.
          When a forceful person forced a driver to drive a
          black taxi to Whitehall.
Responsible:
Target: Cabinet Office.
Damage: Human: Nobody.
        Thing: A black taxi.
Source: telegraph
Source_date: 31 October 1992
Certainty: Facts.
Relevant Information
         Police evacuated 10 Downing Street.
```

Figure 2.3: Example of the contents scanning task.

In the LOLITA contents scanner, the input text is parsed and semantically analysed in order to build a representation in the semantic network. A domain dependent module then searches the network for information relevant to each of the slots. This information, in the form of semantic network nodes, is then passed to the realiser which produces the output.

Chinese tutoring

LOLITA has been used as the core engine for a system to aid the teaching of Chinese to English-speaking students [WAN 92]. One of the main problems encountered

in the learning of foreign languages is the influence of the mother tongue, known as *negative transfer* [SEL 69]. This is the use of native language rules or patterns which lead to an error or inappropriate use in the target language.

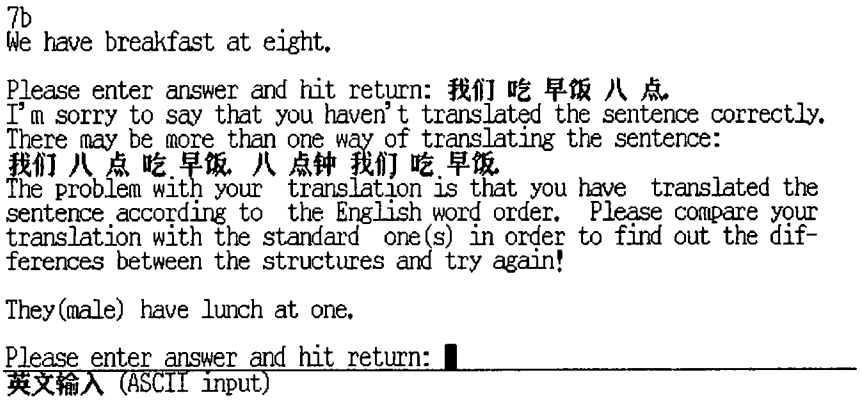


Figure 2.4: An example output from the Chinese Tutor

The Chinese tutor makes use of the technique of a *mixed grammar.* When the parse of a Chinese sentence fails, a partial English grammar rule is invoked for the next part of the sentence. A backtracking algorithm is used in order to cope with incorrect parsing.

The mixed grammar of Chinese and English has been modelled in a way which allows the parser to locate complicated transfer errors not only by examining the error itself, but also by checking its links with other constructs in the sentence. Moreover, the grammatical rules in the mixed grammar of Chinese and English can be used to pinpoint arbitrary transfer errors made by students, without predetermining where the errors might occur. The student can then be informed of any errors, and the system can recommend remedial work.

2.2 Functional Programming

The LOLITA system is written in the functional language Haskell [HUD 92]. Functional languages are a subset of the declarative group of programming languages.

These differ from the procedural languages like C, Pascal or assembler, in many regards. In a procedural language the code takes the form of a list of instructions to be executed sequentially, variables are allocated storage in memory, which can be written to and then read later by a different part of the code. The Declarative approach attempts to allow the programmer to concentrate on problem solving and not have to worry about issues such as memory allocation and sequencing. The reader is directed to [SEB 93] for a more detailed description of programming paradigms. There are three types of declarative languages available to the programmer;

- Specification – for example Z and VDM. These are languages used not to program, but to specify in a precise, and mathematical, manner the behaviour of a system. For instance in Z [POT 91] the specification for a function square is:

$$\begin{array}{|l} square : \mathbb{Z} \rightarrow \mathbb{N} \\ \hline \forall x : \mathbb{Z} \bullet square : x = x * x \end{array}$$

  This states that the function takes as its argument an integer and returns a non-negative number. Furthermore, it also states that *squares* returns $x * x$ (i.e. x^2) for all x that are members of the set of integers. The important thing to note is that this specification does not say how square is to be calculated, but merely what the behaviour of square is, given a valid input.

- Logical – for example Prolog [CLO 84]. A logical language is one in which relations between objects can be defined in terms of axioms, and a query applied to these axioms. For example, given the set of statements:

```
goes_to (boffiness, labs).
owns(boffiness, computer).
studies(H, computing) :- owns(H, computer), goes_to(H, labs).
likes(X,Y) :- studies(Y, computing).
```

and the query:

```
?- likes(boffin, boffiness).
```

Prolog will respond with the answer 'yes.'

- Functional – for example Haskell, Miranda, ML and LISP. The family of functional languages can be categorised by the features identified in Section 2.2.1.

2.2.1 Features of Functional Languages

Lazy evaluation

Not all Functional Languages are lazy (for example ML), however it is an important feature of the language used in development of LOLITA. Lazy evaluation is a method whereby the program will only carry out as much processing as is necessary in order to complete a task. For example,

A function to see if it was nice weather may have the form:

```
IF (dry AND warm AND not-cloudy AND not-windy) THEN good-weather
ELSE bad-weather
```

If the weather was not dry then the lazy system would, in effect, say "we don't need to check the remaining premises as the conditional statement can not possibly be true now, therefore it must be bad-weather". The eager system, on the other hand, would continue to check the other conditions before reaching the same conclusion.

Another result of laziness is that it allows the programmer to generate long, and possibly infinite lists. The only elements which will actually be paid for, in terms of computations, are elements which are used. For example a function to generate the first five prime numbers:

```
fivePrimes = take 5 primes
primes = filter prime [1..]
```

Where prime is a function which produces the value True if the number is a prime and False if it is not.

The function primes takes the infinite list [1..] which is shorthand for the list of integers [1, 2, 3, 4, 5, 6, .. ∞], filters this list to leave only prime numbers and returns a list [1, 2, 3, 5, 7, 11, .. ∞]. The function fivePrimes takes the first five elements of this list, i.e. it returns the list [1, 2, 3, 5, 7]. Clearly if the function primes was not lazy, it would have to generate the entire list [1..] which is impossible. One application of this which will be described later is that of generating a list of possible corrections for a word. It is not necessary to worry about how many of these corrections we can afford to produce, as only the elements of the list which we need are actually produced.

Purity

A language is said to be pure if it does not allow its functions to exert side-effects. A side-effect is caused when the value which a function returns depends not only on the parameters passed to it, but also on the state of some other external variables. A pure language forbids the use of global variables and insists that all variables which are used by the function are passed explicitly. The advantage of purity is that the function application becomes deterministic, *i.e.*, given the set of input parameters, the output will always be the same. This makes integration of new functions, and the upgrading of existing functions, much more straightforward as their effect is limited to the parameters which they are passed, and therefore can not update data being used by other parts of the system.

Abstract Types

The use of abstract data types (ADTs) is by no means unique to functional languages. An ADT is a data type whose representation is hidden from the rest of the system; the ADT provides a set of functions which can be used to manipulate the data type. For example the programmer may wish to make available the data type *Stack*. The ADT may specify the following functions which are permitted on *Stacks*:

- push – adds an element to the stack.
- pop – removes an element from the stack.
- isEmpty – tests to see if the stack is empty.
- emptyStack – creates an empty stack.

This list of function definitions is known as the ADT's *interface*. The programmer is then able to manipulate stacks without being aware of how the stack is implemented. More importantly, should the way in which stacks are represented internally be changed the interface can remain the same and the other developers will not need to alter a single line of their code. In a large system it is highly desirable to use ADTs wherever possible in order to hide complexity and to ease program development.

2.2.2 Haskell

The language in which the LOLITA system is written, Haskell, is a pure, functional programming language with non-strict (lazy) semantics and a polymorphic type-checking system. It was developed following a conference in 1987, as the definitive Language of its type. Haskell incorporates all of the features described in Section 2.2.1; the impact of these features [MOR 94] will be described in more depth during the discussion of the implementation in later chapters. LOLITA was originally

written in the similar language Miranda [HOL 91], but was converted to Haskell in 1993 as this was seen to offer a number of advantages, not least a compiler instead of an interpreter.

2.3 Natural Language Engineering

The LOLITA system has been developed according to the principles of Natural Language Engineering (NLE), rather than ones from the traditional field of Computational Linguistics. NLE attempts to utilise sound engineering principles in the environment of Natural Language Processing research. This pragmatic approach to development means that we do not need to wait for a complete linguistic theory to be developed before we can build a large, realistic and useful NLP system such as LOLITA. Instead we make use of what tools are available to us at present, be they long-standing, well-worked and general theories from computational linguistics and logic, or more localised theories (which despite being unable to cover global possibilities are sufficient to handle what is required), knowledge based approaches, individual heuristics and adaptive or evolutionary techniques.

Since NLE shares a number of factors with more traditional engineering disciplines, these factors have strong influences in the way in which the system is developed:

- **Scale** — The system must be large enough to cope with the demands placed upon it, for example the dictionary must contain a large enough vocabulary.
- **Integration** — The system components should be written in such a way that they are easy to combine with the system as a whole. Furthermore, parts of the system should not make unreasonable assumptions about other parts of the system.
- **Feasibility** — The system should not make unrealistic demands of the hardware platform on which it is executed. The LOLITA system currently requires

the use of a Sun SPARCstation 4 with 64Mb of primary memory.

- **Maintainability** — The system needs to able to be maintainable, in order to ensure its usefulness in the long term. Issues pertaining to maintenance in LOLITA are discussed in [HAZ 93].

- **Flexibility** — The system should be flexible enough to allow its application to other related domains, as need arises. The way in which the LOLITA system can be adapted, through the use of domain specific modules added to the core system, was demonstrated in Section 2.1.1.

- **Usability** — The ultimate aim of the development of any system is to provide a workable solution to a perceived problem. Such a solution must be user friendly in order to be of practical use to an end-user.

- **Robustness** — The problem of robustness is a key factor which any real-world NLE system must overcome. For example, the system needs to be able to deal with ill-formed input without coming to a complete halt.

2.4 Summary

In this chapter the three major factors influencing the design and development of the dictionary and associated spelling correction system have been introduced.

The LOLITA system was presented in order to put the work into a wider perspective. The application of contents scanning was shown, as this is the domain in which the spelling correction system will be particularly useful. Chinese tutoring was also shown to give an insight into the wide range of applications to which LOLITA may be put. Throughout the development of a system it is important to remember what use it will be put to, in particular the concept of a semantic network will be referred to throughout the rest of this work.

Functional programming was described, and a number of features of the language Haskell were introduced. Later chapters will show how these features have

been utilised in order to provide maximum functionality at a minimum cost in both developer and processor time.

Finally in this chapter, the methodology of Natural Language Engineering was introduced. This approach to systems development has influenced the work presented in this thesis by providing a framework of principles to follow.

Chapter 3

Related Work

In this chapter work in the related fields of spelling correction and dictionary access are summarised. The different methods which have been applied to each of the problems are compared.

The problem of spelling correction ([POL 82]) can be separated into two quite different sub-problems. The first task is to detect that a string does not form a valid word, and the second to suggest which word was originally intended.

A great deal of this work has been carried out in the area of text recognition, which requires a high degree of accuracy in error recovery due to the high rate of residual errors [HAN 76]. An optical character recognition (OCR) system [HAR 72] will scan a piece of text, printed or hand written, and produce an electronic version of the original. There are two ways in which errors arise in this system, firstly there may be errors in the original text, or secondly errors may be made in reading the text. These reading errors are generally caused by confusion between one character and another, resulting in character substitution. The other area in which these techniques are utilised is text processing, for example in word-processing or database interaction, where the errors are of a much more random nature. The errors may be due to a lack of knowledge of the correct spelling (convention errors), or due to carelessness or inattention of the author (slips) [HOT 80].

3.1 Detection of Errors

In all spelling correction systems the first task is to find the errors in the text. There are two methods commonly adopted to detect errors, using *n*-grams or alternatively using dictionary lookup. Errors, in this case, are strings which exist in the document being processed which are not valid words. A problem arises with words which are in the dictionary, but not valid at that point in the text. An example would be the wrong use of the words 'there' and 'their', the sentence "I used to live their" contains a spelling mistake but does not contain an illegal string.

3.1.1 *N*-grams

The use of *n*-grams is most common in OCR. An *n*-gram is a sequence of *n* letters which occur in a word. For example, the word 'spell' contains the bigrams *sp, pe, el* and *ll*, and the trigrams *spe, pel* and *ell*. Each string in the input stream is checked to see that it only contains legal *n*-grams. This utilises the redundancy in English of large groups of *n*-grams. The bigram *jz*, for instance, does not occur in any English word, therefore any string containing this bigram must itself be illegal. Leon Harmon [HAR 72] claimed a 42% bigram redundancy, which lead to a 70% chance of a random substitution producing at least one new illegal bigram. In a standard application a 26x26 element array would be set up, Boolean values would be stored in each element, corresponding to the legality of the bigram used to access it. If the word contains a bigram which returns 'false' then this word is flagged as misspelt.

The advantage of this system is that it is light on resources, a 676 element binary array is obviously more compact to store, and faster to access than a 20,000 word dictionary. The main disadvantage of this approach is that a word such as *bal* which is not in the English language, but contains no illegal bigrams, will be allowed.

3.1.2 Dictionary lookup

Dictionary lookup is the more advanced of the two error detection techniques. In principle it would appear a simple task: a string is looked up in the dictionary, if it is there fine, if it is not then flag the word string as misspelt. However the problems arise when one starts to consider the practicality of looking in the dictionary. Clearly if a dictionary contains tens, perhaps hundreds, of thousands of words the problems of dictionary organisation and access strategies begin to look non-trivial. The issues and solutions to dictionary organisation will be discussed in more detail later in this chapter.

3.2 Correction of Errors

Once an error has been discovered the next step is to try to find a plausible alternative string. It is to be hoped that this string will be what the user originally intended the input to be. There are two types of correction algorithms, those which operate on an isolated word, with no thought to its context, and those which use contextual information. For example, the former may correct 'I ate my un' to 'I ate my run', whereas the latter may more sensibly suggest 'I ate my bun'. The use of context here to resolve a choice between possible replacement words is an obvious advantage, but at a cost in terms of complexity. Kukich [KUK 92a] states that work needs to be done to detect 'real word' errors, which she sees as accounting for 40% of all errors. In her experiments she found that the best correction system achieved a 75–85% success rate, and that since only 60% of errors would be identified as such, a spelling correction system would only be able to correct 45–51% of errors.

3.2.1 Correction of Errors in Isolated Words

The correction of errors in isolated words requires the system to give the user a list of possible corrections, based upon the similarity between the initial string and new candidate. A number of techniques exist to generate this list of possibilities. An experimental comparison of some of these methods was done by Karen Kukich [KUK 92a] in the domain of the telecommunications network for the deaf.

Edit Distance Approach

Damerau [DAM 64] puts forward a system for searching through a dictionary looking for a word which differs by less than a set amount from the input string. For example the system may look for a word within one error of the target. The string 'wold' is one error (a substitution in the third letter) away from 'word'. This type of system makes use of the statistics, which Damerau himself gives, that 80% of errors are within one error of the intended word. If Damerau's figures are correct, therefore, a system based on this approach would be expected to have a correction rate of 80%. The method which Damerau uses involves finding a string which is misspelt and then testing every word in turn to see if it is within one edit of the target string. A list would be produced which contained all of the words within one error. An improvement to the algorithm is provided if the dictionary is partitioned according to word length. Each of the simple errors can alter the length of the string by no more than one character. It is sufficient, if looking for single error corrections, only to search the dictionaries containing words within one character of the length of the initial string.

The technique was designed at a time when serial access to dictionaries was inevitable. A serial search of the dictionary would be seen to be very inefficient now that direct access storage media are the norm: the number of comparisons which need to be made for each word is equal to the number of entries in a dictionary.

An example of the use of such a method is systems programs [MOR 70], in

which misspelt keywords in a programming language were corrected. This system, which is described in more detail in Section 3.2.2, has the property of having a small lexicon to match against. In this domain the number of words which have to be looked at is so small that more complex algorithms, in particular the 'reverse minimum edit distance' technique described below, would have to test more hypotheses than there are words in the dictionary.

An alternative to the method described above, which also is based on edit distance, is the so called 'reverse minimum edit distance' technique. This involves generating all of the possible strings within a certain edit distance and testing each of these for dictionary membership until a match is found. This technique was used by Durham *et al.* [DUR 83] for spelling correction in a human interface, and in the SPELL program on the DEC-10 computer. Peterson [PET 80] describes the DEC-10 SPELL program, an outline of which is given below.

```
For each token not in dictionary do
Begin
  Generate list of all strings within one error of target
  Test strings for dictionary membership
  Present user with word(s) to choose correction from
End
```

The main problem with this approach is that a large number of potential solutions are generated and tested. The word 'xxylophone' would produce a very large number of hypothesised spellings which begin 'xx'. These words, would all require checking against the dictionary when clearly, to a human, they can't possibly be correct since no word can start 'xx'.

N-gram Approach

As was seen earlier in this chapter an *n*-gram is a sequence of *n* letters which occur in a word. These *n*-grams can be exploited in spelling correction as well as detection. Each word is seen as a list of *n*-grams which are the features of a word.

A matrix is formed by the features present in a dictionary, and a vector for the string. Vector distance can be used to find the closest match using, for example, dot product. This technique is evaluated in [KUK 92a].

Angell *et al.* [ANG 83] proposed a similarity measure based upon the use of trigrams. They call this the 'well-known dice coefficient', and it is calculated in the following way:

$$2c/(n + n')$$

Where c is the number of trigrams common to each word and within one position of each other in their respective words, and n and n' are the number of trigrams in the two words. Their method involves finding the word in the dictionary which maximises the value of this equation. The main failing with this approach in general is seen to be the very large cost of a serial search of the dictionary. He gets around this problem by using an 'inverted file search procedure'. For each trigram present in the dictionary a list is built of pointers to each word containing that trigram. Therefore, only the words which are referenced in the relevant lists need be considered as potential best-matches. The system can give the desired correction, uniquely, 76% of the time, rising to over 90% if other possibilities are considered.

Ullmann [ULL 77] proposes that in order to make spelling error correction practical it may be necessary to use special purpose hardware instead of a general purpose computer. He goes on to describe a method to process n-grams in parallel. Improvements in computer design over the two decades since the paper was published mean that his arguments relating to execution time of spelling correction programs are not relevant to current hardware platforms.

Rule-Based Approach

A rule-based approach to spelling correction utilises a set of rules constructed to mimic the human way of producing the errors in reverse. Yannakoudakis and

Fawthrop in a pair of papers, [YAN 83a] and [YAN 83b] first present a set of rules for spelling errors, and then derive an algorithm based upon these rules. They assign scores to each word, based upon its likelihood of having been intended, and then the word with the best score is chosen. They use a method of classifying their rules according to three categories: sequential, vowel and consonantal.

Similarity Key Approach

Similarity key [POL 84] methods try to transform the string into a key such that similar strings will map to the same key. The Soundex algorithm, described in [HAL 80], reduces strings to a code of one letter and up to three numbers. It has been used for identifying names in airline booking systems, and in hospital patient records. The first character in the code is the first letter of the word. The rest of the letters in the word are assigned numbers according to the following scheme:

```
0 - A E I O U H W Y
1 - B F P V
2 - C G J K Q S X Z
3 - D T
4 - L
5 - M N
6 - R
```

Any runs of numbers are replaced by a single digit, and any zeros are removed. For example name 'Dickson' becomes 'D022205' which becomes 'D25', the same key as would be generated from the name 'Dixon.' Hall points out, however, that the system is not perfect: the names 'Rodgers' and 'Rogers' are assigned different codes, and there is a problem associated with words which sound alike but begin with different characters.

Pollock and Zamora [POL 84] propose the SPElling Error Detection / COrrection Project (SPEEDCOP) system, which attempts to map similarly spelt words to

the same key. SPEEDCOP uses similarity keys to compare strings to valid words in the lexicon. Two keys are used in the SPEEDCOP system:

1. The skeleton key – the unique consonants are placed in order of occurrence, followed by the unique vowels in order of occurrence. The idea behind this is based upon four principles: (i) the first letter typed is likely to be correct, (ii) consonants carry more information than vowels, (iii) the original consonant order is mostly preserved, and (iv) the key is not altered by doubling or un-doubling of letters or most transposition.

2. The omission key – the skeleton key was found to be lacking for early omitted letters in a word. The omission key sorts unique consonants into reverse order of their statistical probability of omission and then adds the vowels.

The incorrect word is transformed into its key and then all of the words matching that key in the dictionary are listed. Once the candidate words have been retrieved they are ranked according to a plausibility rating based on reversing the error to get to the target and seeing which was the most likely word.

Probabilistic Approach

Probabilistic methods attempt to make use of statistical information in order to find the word which has most likely been corrupted into the string under examination. Probabilistic methods have been used in the OCR environment where confusion matrices are available for characters. For example it may be that the character 'l' is confused with the character 'i' 10 % of the time, and the character 'j' 5 % of the time. Should the correction algorithm come across an 'l' there is a chance it may be an 'i' and a lesser chance it should be a 'j'. The correction is then guided by these probabilities in building up possible target strings. Kukich [KUK 92a] points out, however, that these probabilities are not consistent across applications. It is necessary to train the system, in order to build up a confusion matrix based upon a particular device reading a particular typeface.

3.2.2 Correction of Errors Using Context

It is fundamentally important to first define what is meant by context. Context is taken to be the examination of the word in terms of the words which surround it, at a sentential level or above. This excludes those who consider the context of characters within words (character level), as described earlier. This use of the same word to mean vastly differing concepts can lead to confusion.

Context in spelling error correction can help greatly to resolve ambiguity between hypothesised words, and to find real word errors. Consider the string 'tran', a spell checker may propose the word 'train' as the most likely candidate. If a human read the sentence 'you must be sure to eat plenty of tran', they would suggest 'bran' as the candidate. Clearly the sentence 'you must be sure to eat plenty of train' is nonsense, we use the context in which the word has been used to choose from a list of potential words. It is precisely this use of context which would allow the jump from interactive spell checking, asking the user to resolve ambiguity, to automated spell checking involving the user only at times of uncertainty. Such uncertainty may arise in the following sentence, 'I boarded the tran at the station' (train or tram?).

Morgan [MOR 70] developed a system to correct spelling in computer programs. As was mentioned previously, he used an edit-distance approach to find alternatives, however he uses contextual information to assist in this search. A property of computer languages is that they have a precise syntactic structure. Only a small set of keywords can be used at a particular point in a program. An example of this would be in the line of Pascal code,

IF X = 20 THE Y := 10

The rules governing construction of such a statement would cause the syntactic parse of the statement to fail as it cannot make a valid parse tree from this input. Morgan's system would take the string 'THE' and compare it to the keywords which *are* allowed at this point in the statement 'THEN, AND, OR' a comparison

will be made between the actual string and the list of valid strings, and the most similar will be chosen.

The system enjoys the luxury of a very rigid syntactic structure, coupled with a small vocabulary, quite the opposite of a natural language application.

CLARE [CAR 92] is an NLP system being developed at SRI Cambridge. It uses a lattice-based approach to representation of a sentence. Central to this method is a lattice of overlapping word hypotheses, in which each path through the lattice represents a possible sentence construction. This is similar to the well-formed substring table method of representing multiple possible parses of a sentence.

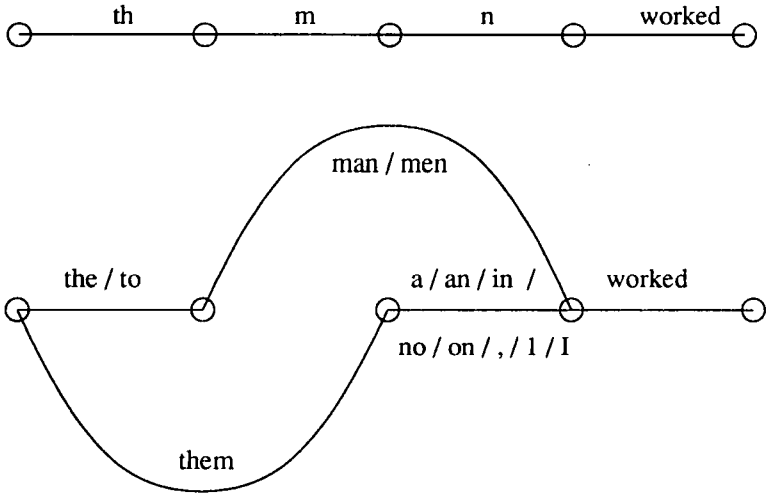


Figure 3.1: A lattice in CLARE.

Figure 3.1 shows the example lattice given in [CAR 92]. The input 'th m n worked' is transformed into the lattice containing possible sentences. Syntax and semantics can then used to filter out impossible routes through this lattice, and the remaining paths are represented in quasi-logical form. If more than one possible sentence results the user may be asked to disambiguate before further processing takes place.

3.3 Dictionary organisation

It has been shown previously that the issue of dictionary organisation is of fundamental importance in consideration of the spelling correction problem: the efficiency of look-up will influence the speed of the system and the structure can influence the implementation of the search strategy.

In the days of magnetic tape this problem was not a factor (or at least one which it was possible to overcome) as access was always serial so a dictionary was literally a word-list. Early spelling detection algorithms were developed with this huge limitation in mind. One strategy to overcome this problem was to sort the words in the file to be checked into alphabetical order and perform a serial comparison between the dictionary and the ordered list of strings. The serial comparison method allowed for dictionary membership to be tested but the speed overhead made postulating valid alternatives a difficulty. Modern technology has greatly assisted the dictionary lookup problem, it is now possible to directly access a backing storage device. Indeed the primary memory in some workstations can be large enough to hold even a large dictionary, reducing the cost of a dictionary access. [KNU 73] provides a detailed discussion of data-structures.

3.3.1 Hash Tables

Hash tables are the most common form for dictionary storage, a hash value is calculated by applying a function to the string. For example the hash function may take the ASCII values of the letters in a word, add a weight to each of them dependent on their position in the word, and then sum these values. This value will be used to access the word in the dictionary. This address may contain a 'true' flag to show that a word exists, or the word itself. There will, of course, be problems in this system if it is possible for two strings to map to the same address. In the boolean system this may result in an incorrect string chancing upon a valid hash address and being passed, if the word is stored we have the problem of what

to do if two words map to the same address. There is a trade-off to be made between the size of the hash table being large enough to avoid the type of problem described above, and being small enough that it does not use large amounts of system resources for storage of unused addresses in a sparsely populated table. It does have the advantage, however, that the random access nature of accessing the table eliminates the need for comparisons in other techniques. The pseudo-random distribution of words mean that closeness of hash value, and therefore positioning in the table, has often no relevance in terms of word similarity.

3.3.2 Binary Trees

For a small dictionary an ordered binary-tree representation may be an efficient representation. A binary tree, as shown in Figure 3.2, is the data-structure equivalent of a binary search.

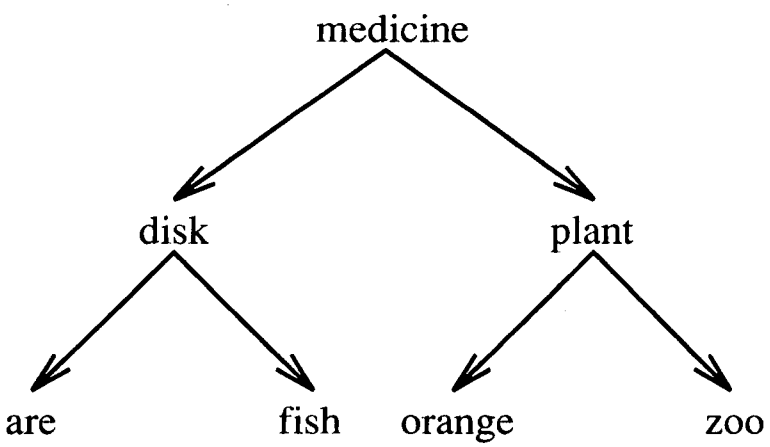


Figure 3.2: A balanced binary tree with seven entries.

The dictionary is organised so that the median word in the dictionary lies at the root node. A comparison is carried out to see if the value of the string which is being looked for is greater or less than this word. If the value is less then a branch is made to the left, if larger to the right, until the values match meaning the word has been found. If the bottom of the tree is reached, meaning it is no longer possible to branch, the search is said to have reached a leaf node, meaning that the word is

not in the dictionary. The tree only maintains its binary search properties as long as it is balanced, that is at each node the number of nodes reachable to the left, and the number to the right are the same, plus or minus one. If a node is added, or removed, it is necessary to rebalance the tree, to make sure that no branches of the tree are longer than the others by more than one node.

3.3.3 Tries

A trie [FRE 60] is defined as:

> An M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level χ represents the set of all keys that begin with a certain sequence of ε characters; the node specifies an M-way branch, depending on the $(\varepsilon + 1)$st character.

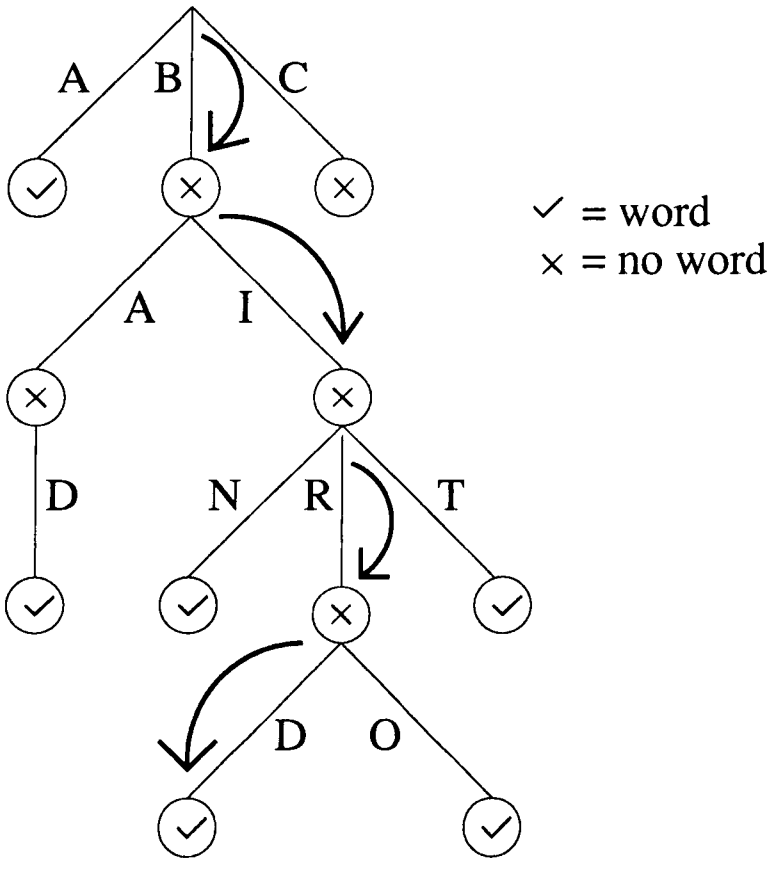


Figure 3.3: Finding *bird* in a trie.

Figure 3.3 shows how the words *a*, *bad*, *bird*, *biro*, *bin* and *bit* could be represented in a trie structure, and the way in which the word bird is accessed.

The word trie is taken from the structures main use, information re*trie*val ([KUK 92b]). A trie dictionary is, in fact, a large finite state machine through which there is a path to each word in the dictionary. If the string which is being looked for in the dictionary does not follow an existing path then that string can not be a valid word. Furthermore, if a string ends at a position in a path which is not marked as being a word, that string is rejected also, it being the starting substring of a valid word, but not a word itself. The main advantage of tries is seen to be the independence of lookup time for a word from size of dictionary. The number of nodes which need to be traversed to reach a word is related only to the length of the word.

This theoretical independence, however, is not fully realised when implemented. The time taken to classify a string as not a word will increase, because it is more likely there will be a path down which it can partially fit. In addition, as the number of branches possible at each node increases so will the time taken to select the correct one. A number of techniques have been proposed for compacting trie structures. Comer [COM 81] provides a heuristic for minimising tries in which all the leaves lie at the same level, clearly not the case in a dictionary system. Ramesh *et. al.* [RAM 89] and Al-Suwaiyel and Horowitz [ALS 84] both present methods for the optimisation and compaction of trie structures.

Dunlavy [DUN 81] describes a spelling correction system which called SPROOF. SPROOF makes use of a trie dictionary and heuristic search, in order to find words within a set edit distance from the string under examination. Furthermore he claims, using a mathematical model, that his algorithm exhibits 'good efficiency even at dictionary sizes of 2^{96}'. Muth and Tharp [MUT 77] have developed a dictionary system which implements a trie using pointers. Each record, or node, in the trie contains the letter which the node represents, pointers to its father node (one level up), a brother node (same level) and a son node (one level down). To find a word in the dictionary from the top node, one follows the son node, if the

entry in this record is the letter which is next in the word then repeat, else move on to the brother and try that record.

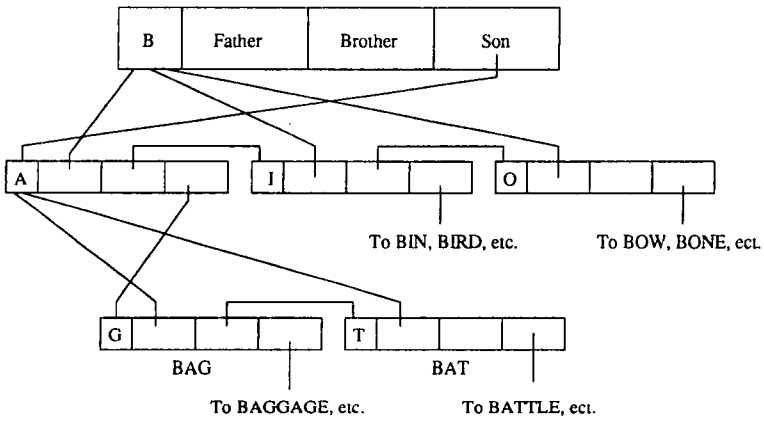


Figure 3.4: A section of Muth and Tharp's trie.

Figure 3.4 shows how a portion of a trie looks under Muth and Tharp's representation. In order to find the word 'bat', traverse down the son arc from the 'B' node, this node represents the string 'ba', as this the starting string which is being looked for we can traverse the son arc again. At this node the word 'bag' is represented, this is not the third character being looked for, so the brother node is traversed. The traversal of the brother node leads to the node which represents 'bat', the word is found.

Downton [DOW 82] made use of a trie to store palantype chords and the English words which they represent. A palantype is a mechanical shorthand machine. Downton developed a system to transcribe from the shorthand produced by a palantype into English for a deaf person to read. The chords, in this system, are treated as letters would be in a standard implementation, and the English words stored at the nodes. He notes that this type of storage is "substantially superior" to other search methods he had investigated.

3.3.4 Dictionary size

There is some debate about the optimal size of the dictionary. There is a clear trade-off between having a lexicon large enough to contain as many valid words as possible, to reduce rejection of valid words, and the conciseness of a lexicon to reduce the risk of errors creeping in as they are passed off as other words. Peterson [PET 86] argues the case for a small dictionary stating that with a large word list almost one in six typing errors will be passed off as other words. Damerau and Mays [DAM 89] claim that in their experimentation the increase in size of a dictionary from 50 000 to 60 000 words reduced the overall rate of misclassifications by between 50 and 150 times. This argument, however, is somewhat irrelevant in terms of LOLITA where the number of words in the dictionary is as large as possible in order to ensure maximum coverage of the English language.

3.3.5 Dictionary partitioning

As was described earlier, some of the work which is done by spelling correction systems can be reduced by considering only sections of the dictionary which may contain the word being looked for. In addition, the partitioning of the dictionary into smaller sections will reduce the loading time for the dictionary, if it is only partially used. A discussion of dictionary partitioning is given in Section 4.2.3. The DEC-10 SPELL algorithm [PET 80] partitions the dictionary into 6,760 sections. The dictionary is partitioned by the first two letters, and by length. Peterson himself envisaged a dictionary split into three, as shown in Figure 3.5. He claims that over half of the words used in English come from a vocabulary of 136 words. Furthermore, he feels that by building a second dictionary of words used already in the document being checked, the need to resort to searching the large, and therefore slower, dictionary will be reduced.

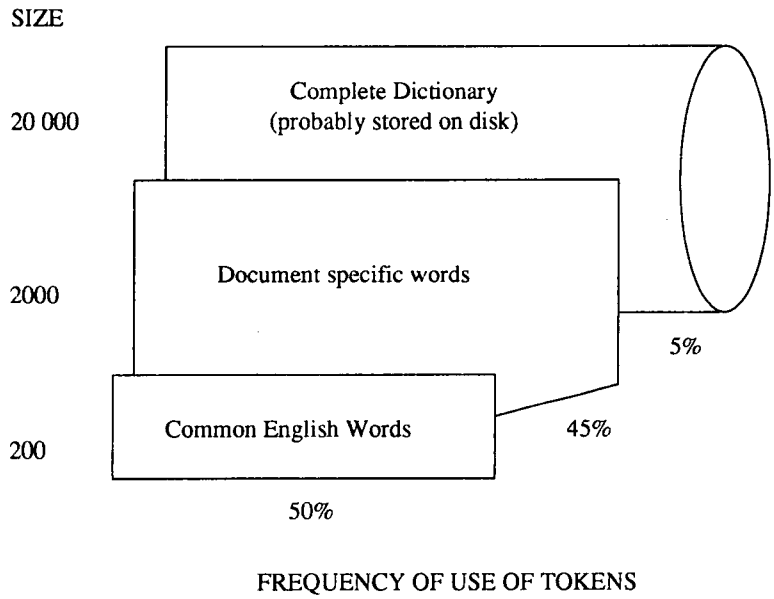


Figure 3.5: Peterson's dictionary partitioning scheme.

3.4 Summary

In this chapter a number of methods for detecting and correcting spelling errors have been introduced. The two approaches to the detection problem were seen to be the use of n-grams and the use of a dictionary against which to check the strings. The use of n-grams was seen to be the least expensive in terms of storage and of computations. However, the accuracy of the n-gram approach is not as good as that found when using dictionary access. Spelling correction is usually done in the in isolation, i.e. no contextual information is used in order to assist the search, or to disambiguate between alternatives. One major shortfall of systems which do not use any contextual information is that real word errors are missed altogether. Again the distinction between n-gram based approaches and dictionary based approaches was apparent. Some more novel solutions to the problem were also discussed, such as the similarity key approach. A number of techniques for storing dictionaries were also introduced. Clearly, the method of dictionary storage which will be employed is largely dependent on the way in which it is to be used. The issue of dictionary storage was also seen to involve a number of choices, quite

separate from the data structure itself, namely size and partitioning decisions. The next chapter describes the development of the dictionary system for use in the LOLITA NLP system, and Chapter 5 that of the spelling correction system.

Chapter 4

The LOLITA Dictionary

The LOLITA system uses dictionaries for the three languages to store the root forms of words and one or more pointers to the node(s) in the semantic net which it represents. Each word may actually have a number of meanings, the word "point", for example, has eighteen references to nodes in the semantic net.

In this chapter the development of a new storage structure for LOLITA's dictionaries will be presented. The choice of a data structure is described and the design and implementation of the dictionary is outlined. The use of profiling tools is introduced as a method of 'tuning' the data-structure for maximum efficiency. Finally, the testing carried out on the new dictionary is presented.

4.1 Selecting a data-structure

The choice of data structure used to store the dictionaries in any system is a trade-off between a number of factors.

- Functionality – The use to which the dictionary is to be put is of fundamental importance. If the dictionary is to be used, as Damerau envisioned [DAM 64],

to check against an ordered list of strings for dictionary membership, a serial word list may be sufficient. If, however, more sophisticated functionality is needed, for example the spelling correction described in Chapter 5, it will be necessary to adopt a more complex data structure. There is also a need to specify what the most critical use of the dictionary is. It may be possible to create a structure which takes a long time to load and build but is very fast to access. The places in which savings are to be made will greatly affect the choice of structure to be used.

- Development Environment – Central to the design of any system must be the limitations placed upon the developer by the programming language being used. A system which is reliant on fast array access would not be suitable if the language being used did not support arrays. Equally, the concept of lazy data structures introduced in Section 2.2.1 provides the developer with a powerful programming tool.

- Space – The discussion of space centres around the two types of storage available to the modern computer user: disk and primary (RAM) storage. The environment in which the dictionary is to operate may allow limited use of primary memory, in which case it will be important to design a structure which does not necessitate the loading of the entire dictionary from disk, in order to look up one word. Alternatively the opposite may be true, the system may be limited in secondary storage, therefore requiring an efficient method of storing the dictionary on disk.

- Time – The response time of the dictionary is an important factor to be considered. In a batch environment where response time is not a factor it may be possible to use a less efficient access technique in order to minimise the storage requirements as described above. If, alternatively, the dictionary is to be used to process text in real-time then a fast access capability is essential. It is necessary to look at the time taken for different operations, for example, saving, accessing and reading, and strike a balance between them.

- Scale – The size of a dictionary will influence the choice of structure greatly. A solution which works very well for a dictionary containing 1000 words may be completely unworkable for a 100 000 word dictionary. In considering issues of scale it is important to look not just at the current system, but also at what future demands may be placed upon the dictionary.

The LOLITA dictionary structure introduced in this chapter has been developed with these factors in mind. The initial motivation for changing the data-structure used within LOLITA was the addition of spelling correction facilities as described in the next chapter. The first attempt at spelling correction involved generating a list of all words within one error of the initial string, and then testing each for dictionary membership. This requires $52n + 24$ lookups for a string of length n (see Section 5.1). This would mean that for the average length string in the LOLITA dictionary, 8 characters, a total of 440 dictionary accesses would have been required. The dictionary structure being used by LOLITA was not able to carry out that level of access in an acceptable time. It was felt that, in order to make a spelling correction system viable, an improvement in dictionary access time was vital.

The first task in attempting to speed up dictionary access time was to investigate whether improvements to the current system could yield the necessary improvements.

In the past, the LOLITA system has made use of a hash table representation for efficient word lookup. The table was originally implemented in Miranda as a doubly linked list. The move from Miranda to Haskell would have enabled the use of updateable arrays, but the system had not been altered to use these as the change would have required a substantial rewriting of the system. The LOLITA hash dictionary was implemented as a collection of *hashchunks*, which are small sections of the hash table; LOLITA's hash dictionary consists of 200 of these hashchunks. A hashchunk, part of which is shown below, is defined as a list of 100 *hashlines*:

```
[("windmill",[257])]
[]
[("stupid",[258,23136])]
[("attempt",[259,21393]),("prey",[260,22681])]
[]
```

The hash table entry for each word contains the word itself and integer references to the semantic network which represent its meanings. It is possible for two words to share a hash address, in which case it is necessary to scan the hashline for the word which we are seeking. Looking up a word involves computing its hash address, accessing the correct hashchunk (scanning on average 50 hashlines) and then, finally, a scan and comparison along a hashline. This had proved to be an acceptable cost when checking the validity of a word given to LOLITA. However, because of increased demands placed upon the dictionary, it was not felt that the old system would be able to provide the level of functionality which could be expected from an alternative structure.

Various alternative dictionary structures were examined, in order to assess their suitability for the task. The spelling correction consideration made it desirable to find a method whereby similar strings were stored in the same part of the structure. This would save loading a lot of different sections of the dictionary into memory in order to look for possible spelling corrections.

- Binary tree – The use of a binary tree to store the dictionary was considered, and rejected. The size of the binary tree needed to support a dictionary the size of LOLITA makes it unworkable. The LOLITA dictionary will contain approximately 45,000 words, in a binary system this would necessitate a 15 level deep tree. The average number of comparisons required to find a word would be approximately ten, a figure too high for this to be a practical solution. In addition to the problem of size there are problems of keeping the tree balanced and the amount of time which would be spent re-balancing the tree after the addition of, or more problematically the removal of, a word

from the dictionary.

- Trie – It was felt that the advantages provided by the trie structure were sufficient to warrant the implementation of a new dictionary system based upon this structure. A trie, as was described in Section 3.3 is an M-ary tree. Access time to the structure is, in theory, constant for words of the same length, regardless of the size of the dictionary. This was an important consideration as the data stored in the dictionary is expanding as the system develops. With work currently in progress which will increase the amount of words stored in the dictionary, clearly a system was required which could cope with this change. The second factor in favour of the use of tries is the advantages their structure confers to spelling correction. The use of the trie structure for spelling correction will be described, in depth, in the next chapter. There are also memory advantages with this structure over others considered. Starting strings shared by more than one word need only be stored once, e.g. the string 'tele' in television, telephone, telemetry, etc.

4.2 Design

There are two areas of design to be considered: the trie structure itself, and the nodes which lie at each branching point and at each leaf.

4.2.1 Tries and subtries

The main design decision to be made at this point is how to store the nodes to which it is possible to branch from a given node. The number of nodes reachable from a node will vary largely, the top node may contain well over thirty subtries leading from it ('a' to 'z' plus other assorted characters which which a word may start), a node near the bottom of the trie possibly one or two, and the leaf (end) nodes will have, by definition, no accessible nodes below. The choice to be made

was between using a list or an array. The use of a list offered the advantage of a saving in memory space, since only subtries which are non-empty need to be kept, if a character's subnode does not appear in the list it is assumed to be an empty trie. The use of a list also offers a reduced setup cost, and, in the implementation of Haskell used for LOLITA, faster access than for arrays if the number of elements in the list is small enough. Arrays offer greater speed of access when the number of subtries contained becomes large enough. However there are two problems: firstly, an element must be initialised for every element of the array, regardless of whether it is empty or not; and secondly, the array would only hold the characters 'a' to 'z', anything outside this range would require the use of an 'exception list'. The range 'a' to 'z' was chosen as, with the exception of a very few words, the characters used in English fall in this range. The use of larger arrays to deal with these relatively rare instances would have resulted in higher access times for all words. Aoe *et. al.* note this problem and provide a solution using what they call *double-arrays*, the reader is directed to [AOE 92] for further details.

Clearly the list vs. array debate is an important one, it was felt that the use of arrays was prohibitively expensive when a low number of elements was to be stored. Similarly, it was felt that the access time for a list with a large number of elements was too high to be useful. A solution was designed which would take advantage of the quick access time of arrays when the number of elements in the list reached a certain threshold. A trie, therefore, can be one of three things:

- Empty Trie – The default value. The presence of an empty trie means that there are no further words which can be reached by continuing along the present path.
- List Trie – Once an element is inserted into an empty trie it becomes a list trie. The list trie takes the form of a list of associations, for example, '*c*' = *subnode*. The list is made up of associations of this form because it is these associations which can be converted directly into an array.
- Array Trie – When an attempt is made to add a node to the list trie which

would take it over a certain limit (to be introduced later), the node list is converted into an array. Because of the exception list, the array trie actually takes the form of a pair of list and array. The format of the list results in the conversion being relatively straightforward, and any element which does not have an entry in the list is given a null node.

4.2.2 Nodes

Once the trie structure had been decided on, the next stage was to decide what the trie should store. The trie shown in Figure 3.3 is the simplest possible structure, providing the minimal functionality required of a dictionary, a test for membership. The requirements of the LOLITA dictionary were slightly more complex. The dictionary acts, as would a conventional paper dictionary, as more than simply a list of words. The entry for each word contains one or more references to points in the semantic net, the equivalent of a paper dictionary containing the meanings of each word. Each node in the trie, therefore, must contain a list of references into the semantic net (noderefs) for the word which lies at that point.

It was also felt that the trie structure could be used to store prefixes and suffixes, for use during morphological analysis. For example the word 'unloaded' is made up of the root 'load' in combination with the prefix 'un' and the suffix 'ed'. These morphological features may be deeply nested, for example the word 'antidisestablishmentarianism', which has as its root the word 'establish'. For this reason, every node in the trie also contains a morphological flag to state if the path followed to reach this point could be a prefix, suffix, both or neither. The final element required at each node is a subtrie, possibly empty, which contains the words which have the characters so far used to reach the node we are at as their opening substring. Figure 4.1 shows how a node in the dictionary is represented.

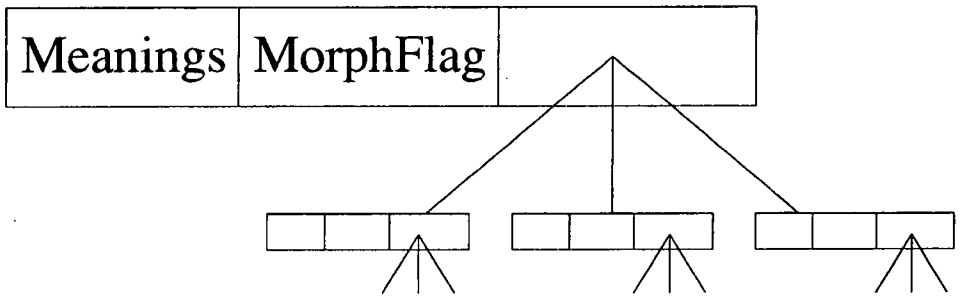


Figure 4.1: Node format.

4.2.3 Dictionary partitioning

The lazy properties of functional programming languages allow the dictionary to be partitioned in order to save time loading from disk. There is a trade-off to be made at this point between splitting the dictionary into tiny fragments so that only the bare minimum of data is loaded for each access, and having large sections in order to cut down on disk access overheads. Each time a file is accessed on disk there is the cost of getting the disk spinning, reading the FAT to find the address on the disk of the file that is being accessed, moving the head to the address and then the cost of reading the file itself. Clearly, if 10Kb of data is read from a disk it will be considerably faster to read it in one go than ten disk accesses for 1Kb.

It was felt that the best way to partition the LOLITA dictionary was by taking subtries after two arcs have been traversed, i.e. all words with the first two characters in common are stored in a file together. The split was made at this level as it was felt to be the best compromise between number of words per file and number of files. Splitting the dictionary one level higher, after one character was not an option because when the spelling corrector attempts substitutions on the first letter of a word, the whole dictionary would need to be loaded. The partitioning after the second letter produces 676 files, to partition after 3 letters would produce 17576 files, which was felt to be prohibitively high. There are, of course, words which do not fit into this partitioning scheme. Words of a length less than three, or which contain characters outside the 'a' to 'z' range, need to be dealt with separately.

4.3 Implementation and Integration

Hazan el et al. [HAZ 93] claim that the use of functional languages in the development of LOLITA eases program comprehension and, more importantly, makes the integration of new code into the system less troublesome. This has proved to be the case during the integration of the new dictionary. In particular the coding of the dictionary as an abstract data type (ADT, see Section 2.2.1) simplified integration. The specification for the dictionary was provided by the interface of the module, this states which functions are provided by the dictionary, and the signature of each. For example, the function `addWordMeaning` which adds a meaning (`Noderef`) to a word:

```
addWordMeaning :: Word -> Noderef -> WordDict -> WordDict
```

States that the function `addWordMeaning` takes as its input parameters values of the type `Word` (the word to add a meaning to), `Noderef` (the meaning) and `WordDict` (the dictionary), and returns a `WordDict` (the updated dictionary). Because the `WordDict` is an ADT the functions which use it have no knowledge of its implementation, as long as the signatures of the dictionary manipulating functions don't change, the rest of the system will notice no difference.

4.3.1 Implementation

In order to present the implementation details, each of the functions in the dictionary's interface will be introduced, along with descriptions of any important features in their implementation.

initWordDict :: [Char] -> WordDict

The function `initWordDict` initialises the dictionary for usage. Given the language to use (e.g. English, Italian) the function creates the base dictionary ready for use.

`initWordDict` only actually loads the 'oddities' which don't fit into the regular file structure, the rest of the words are inserted in the form of subtries at the appropriate nodes. Due to lazy evaluation the subtries are not loaded until they are needed. This means that although the data structure is initialised and ready to use, the loading costs of each subtrie is deferred. This is where the use of functional languages has made this approach practicable, the cost of loading and building the entire dictionary may be unacceptably high using a trie structure in the way which LOLITA does. However the use of lazy evaluation means that once the dictionary has been set up the rest of the system can treat it as if it is all there, Haskell copes with the loading of individual parts when needed. It is important also to realise that once a section of the dictionary has been loaded it is not discarded. The difference between lazy evaluation, and the idea of partitioning the data into completely separate 'mini-dictionaries', is that each portion only needs to be loaded once.

saveWordDict :: WordDict -> [Char] -> [([Char], [Char])]

The saving function is an example of the way in which the LOLITA system is implemented in order to hide abstractions. The function, given a dictionary and a language, returns a list of filenames and contents. This allows the handling of output, a non-trivial task in Haskell, to be passed to a dedicated part of the program which handles all output from the system. When saving the dictionary it is important that the system only saves the portions of the dictionary which have been changed in some way. In order to facilitate this the actual structure of the `WordDict` is a pair containing the trie structure and a list of which parts of the dictionary have been changed. When the dictionary saving routine is invoked the only sections of the trie which are saved are those whose index appears in this list. If this list is empty then there is no need to save anything, as the dictionary in memory is the same as the one on disk. The dictionary files are named according to the section of the trie which they represent, for example, all the words beginning 'ha' are stored in the file 'trieha'. The oddities described earlier are stored in the

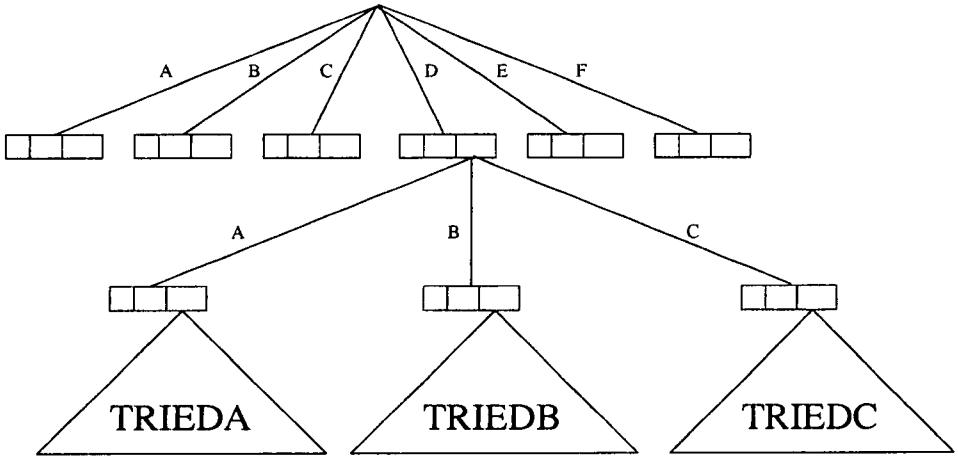


Figure 4.2: A portion of the dictionary showing how the subtries are utilised.

file 'trieodd'.

The files for each language are stored in different directories in order to keep them separate. The format of each file was designed in a way to make it readable by a human so that any problems could be fixed by hand, for example four entries from the file 'triewi',

```
("se",'N',[3615, 23428])
("sh",'N',[18169, 20694, 26206, 67159])
("stful",'N',[1554])
("thdraw",'N',[9637, 26207])
```

The first two letters are omitted since, by definition, all words in the file share the same opening characters. The second member is the morphological flag - in the above example none of the entries represent a morphological feature. The final entry is a list of the `Noderefs` for that word.

forceLoadWordDict :: WordDict -> Int

The property of laziness is not always desirable, there are times when it is more appropriate to load the entire dictionary in one go. In order to do this it it is necessary to specify a function which forces the evaluation of the entire trie (by

definition it must also return a value). This is achieved by counting the number of leaf nodes in the trie. Because every path through the trie must terminate in a leaf the function must visit every node in the trie, therefore forcing the loading of the trie. This capability is especially important when response time is of vital concern, but setting up time is not: the system can spend some time loading in all of the dictionaries and the semantic net so that they are in primary memory when they need to be accessed. It is also often useful to force the loading of data-structures when the efficiency of new modules is being investigated. The deferral of loading costs may make other parts of the system look slow, when in fact they are spending their time doing work which was initiated by other, earlier, parts of the system.

getDictWord :: WordDict -> Wordref -> Word

This function is used to map from the `Wordref` stored in the semantic net, to the word itself for output to the user. In the old system a `Wordref` contained an address in the hash dictionary at which point the word was stored. Strictly speaking, this was a violation of the ADT system: the semantic net depended upon a particular representation being used for the dictionary. There is no way to access a word in the trie dictionary, except with the word itself, therefore the abstract type `WordDict` was changed from a pair of integers to a string, the word itself. This change necessitated the conversion of the semantic net, from containing the old `Wordrefs`, to containing the word itself. This was achieved by allowing both representations to exist, side by side, and iterating through every node in the net changing its entry. It is important to realise that the addition of the words to the semantic net in no way compromises its language independence.

As shown in Figure 4.3, the different language versions of each word are represented as a link from the English version.

The functions which deal with the `Wordref` type are preserved in order to maintain the functionality should the representation be changed again in the future. The function `getDictWord` simply returns what it is given, stripped of its constructor,

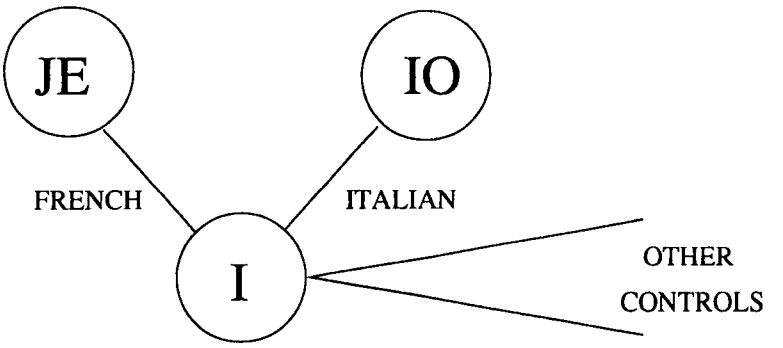


Figure 4.3: A simplified semantic net fragment showing language controls for the word *I*.

e.g. `Wref "fish"` becomes `"fish"`.

deleteDictWord :: Word -> WordDict -> WordDict

The deletion of a word completely from the dictionary is achieved using this function. The deletion of a specific meaning of a word requires use of the function `deleteWordMeaning` described later. In order to delete a word from the dictionary it is necessary only to find the word in the dictionary and replace the list of `Noderefs` at that point with an empty list, and to add the partition to the list of revised partitions. As the saving function is traversing the trie and building up a file to save, it will not save the word as there are no `Noderefs`. The save method has, therefore, eliminated the need to backtrack up the trie, removing the path to that word if no other words exist down that path.

getin :: WordDict -> Word -> [Noderef]

The function `getin` is the most frequently used function for the dictionary. It is the equivalent of looking up a word in a paper dictionary: given a word and a dictionary, getin returns a list of `Noderefs` (meanings). The list of `Noderefs` will be empty if the word is not in the dictionary. It is the execution speed of this function which is central to the overall gain in performance which the conversion to the new dictionary will bring to LOLITA. One of the main reasons for adopting the trie based approach was that the speed of execution of the `getin` function

would not be affected by increasing the size of the dictionary. In particular the addition of spelling correction facilities places a heavy burden on this function when used to rule out hypothesised spelling corrections. In the above discussion on factors influencing dictionary storage, it was stated that it is necessary to find the area of performance which it is most desirable to optimise. The `getin` operation was seen to be the most critical of the dictionary operations, since the checking of hypothesised corrections involves looking up each suggestion in the dictionary – the job served by `getin`. The procedure followed in optimising the `getin` function's performance is described in Section 4.4.

addWordMeaning :: Word -> Noderef -> WordDict -> WordDict

This function is used to add a new word to the dictionary, or to add an extra meaning to a word which is already in the dictionary. The addition of a word to the dictionary requires the traversal of the trie to the node which represents the word, and then the insertion of the given `Noderef`. If the word already exists the new `Noderef` as appended to the list of `Noderefs`. If there does not exist another word with the same starting string then it will be necessary to build the path to the word. Finally the word's index must be added to the list of sections of the dictionary which have been changed, so that when the dictionary is saved the changes will be preserved.

deleteWordMeaning :: Word -> Noderef -> WordDict -> WordDict

It is sometimes necessary to remove from the dictionary not a word itself, but one or more of the meanings (`Noderefs`) associated with it. The function has the same functionality as the `deleteDictWord` function when there is only one meaning for a word in the dictionary, otherwise it removes the given `Noderef` from the list of `Noderefs` for that word.

getWordref :: WordDict -> Word -> Wordref

This function does the opposite of the function `getDictWord` described previously. As with the `getDictWord` this function no longer actually uses the dictionary, needing only to add the constructor to the word it is given. For example `"fish"` becomes `Wref "fish"`.

nullWordRef :: Wordref

This function is simply used as a null `Wordref`, and will always return the value `Wref ""`.

addDictWord :: Word -> WordDict -> WordDict

This function has been superseded by the function `addWordMeanings` and is only present for backwards compatibility. It is used to add a word without a meaning, but since doing this is of no real value its use is discouraged.

4.3.2 Integration with LOLITA

The integration with LOLITA was relatively straightforward because of the use of ADTs, as described earlier. The change in the software itself was accomplished simply by replacing the old dictionary control module with the new one. Since the interface was identical it was not even necessary to compile the entire system, just the module itself, and then linking the modules. The main task to be performed was to convert the four dictionaries and the semantic net to the new format. This was achieved by using the old read functions to provide a list of words and meanings, inserting these into an empty trie, and then using the save function to write the new dictionaries to disk.

4.4 Profiling the dictionary

Profiling allows the developer to investigate the efficiency of code through the gathering of statistics detailing how long the CPU has spent executing the code in each function. These times can then be used to compare the execution of different implementations, to see which is most efficient. The use of lazy evaluation can obscure the execution speeds of sections of the system which spend time performing work deferred from earlier ones. Because the profiler allocates costs associated with the execution of each function, the deferred costs will be allocated to the original function, not the function which requires its evaluation. In addition the profiler can give very accurate figures relating to the execution time of an individual function, rather than using the UNIX *time* command which will only give an overall time. The profiler was used in order to 'tune' the data structure, and so minimise the execution time of the function `getin`. As has been described earlier, the trie structure uses a threshold, τ, at which the switch from using a list representation to an array representation is made. In order to find the value of this threshold, the function `getin` was profiled. The function was isolated, so that all costs of sub-functions of `getin` would contribute to `getin`'s cost. The profiler was then used to give a value for the CPU time to look up every word in the dictionary 10 times - a total of over 200,000 dictionary accesses.

The results, as shown in Figure 4.4, show the optimal value of τ to be three. This value may, however, change with more efficient implementations of lists (τ rises), or arrays (τ falls), in future versions of the Haskell compiler. The shape of the curve is as would be expected. Using an array for almost all of the nodes in the trie introduces a very large cost as there will be a lot of arrays at the bottom of the trie with very few elements. The slower rise in execution speed, as less arrays than is optimal are used, can be attributed to fact that the inefficiencies of list access worsen with size of list. The use of the profiling tool has been an important part of development: as can be seen in Figure 4.4, the correct setting of the threshold value is vital. If the setting is as little as two elements either side of the optimal it can have disastrous effects on the efficiency of the dictionary.

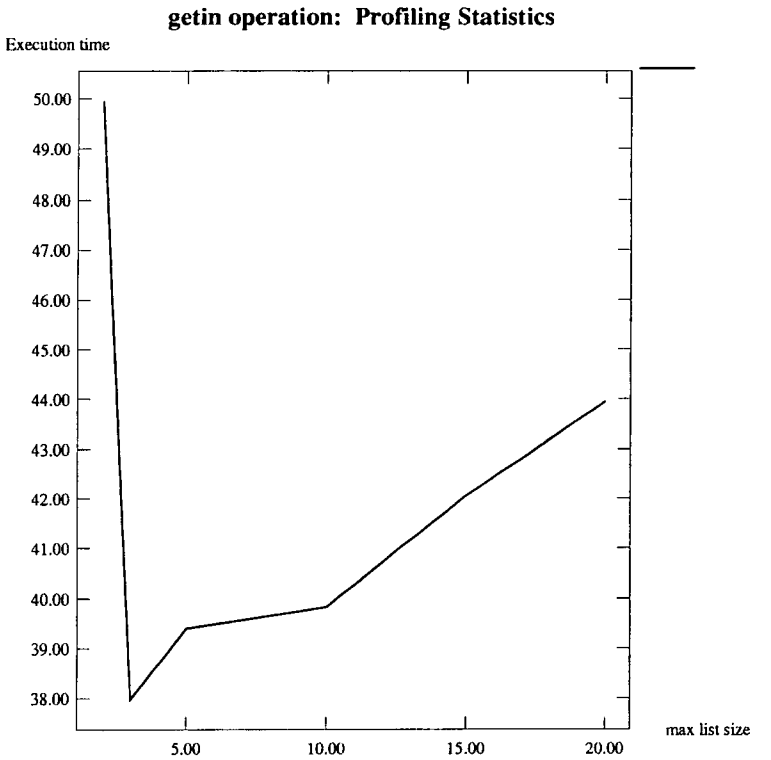


Figure 4.4: Results of getin operation profiling

The profiling of the LOLITA dictionaries is used as a case-study in [SAM 94].

4.5 Testing

The first task in the testing of the dictionary was to run through the demonstration of LOLITA comparing the output of the new version to that of the old. The demonstration of LOLITA was used in order to test the performance of the dictionary, because it contains an example from all aspects of the system. The initial testing of the dictionary revealed a small error when adding a word whose first two characters were not already present, but this bug was rectified and the dictionary has been in use by the twenty LOLITA developers for over two months with no problems.

The overall performance of the dictionary was tested by iterating the demonstration a number of times and comparing the performance of the new dictionary

with that of the old. The new structure was found to have a slightly higher start-up time, however, the faster execution time once the system was initialised meant that this difference was negated during practical use.

Since the implementation has been completed, the size of the dictionary has been increased. The increase in dictionary size has resulted in an inevitable increase in the build and save times, as each subtrie to be loaded contained more words. The access time to the dictionary, once loaded, has not been greatly affected by the increase. Unfortunately, due to the fluid nature of LOLITA, it is not possible to present comparative figures for old/new access times, since the rest of LOLITA is continually changing, affecting overall performance.

4.6 Summary

This chapter has described the development of the trie based dictionary for the LOLITA system. Tries were chosen as they offered the most assistance to the spelling correction system, and because the access time would not be affected by the forthcoming increase in dictionary size. The use of the functional language Haskell has influenced development in two major ways. Firstly, the use of lazy evaluation has assisted dictionary partition by allowing the deferral of load costs until use. Secondly, the use of ADTs, coupled with the concept of purity, has helped make the introduction of the new system cause minimum disruption to the rest of the system.

Profiling was introduced as a helpful tool to gain an insight into the execution characteristics of the code, in this case to find the optimal setting for the list/array changeover threshold. Finally, the dictionary was tested and found to produce favourable results in comparison to the hash dictionary. The real benefit of the new dictionary will be shown in the next chapter, as the spelling correction algorithms are presented.

Chapter 5

The Spelling Correction System

In this chapter the spelling correction strategy is presented. Firstly, the rationale behind choosing this approach over other possible solutions is described. The four error forms are discussed in greater depth, and then the algorithm is presented. Finally, implementation and testing of the spelling correction strategy is described.

5.1 The choice of algorithms

Chapter 3 reviewed the related work in the field of spelling correction. The methods described in Section 3.2 were examined to assess their suitability for the LOLITA system, and the following observations made:

- N-gram Approach – The first choice to make was whether to use an n-gram based system or a dictionary based system. The LOLITA system needs a dictionary in order to store the references to nodes in the semantic network, so the use of a dictionary to detect spelling errors was inevitable; having passed a word using n-gram methods the word would still have had to be looked up in the dictionary to find the semantic net reference. The possibility of using n-grams in order to aid the spelling correction was investigated. Harmon

[HAR 72] claimed a 42% bigram redundancy, which lead to a 70% chance of a random substitution producing at least one new illegal bigram. When the LOLITA dictionary was tested, to determine if this figure would apply to the data set which was being used, a redundancy figure of only 19% was achieved. This figure makes the use of bigrams unrealistic with this data-set.

- Edit Distance Approach - The approach of checking the dictionary in a serial manner and finding the words which have the lowest edit distance was rejected. The time spent looking up every word in the dictionary and hence computing its distance was considered to be too inefficient. The reverse-minimum edit distance technique was examined. The system generated all of the words possible within one error of the input string. In considering the efficiency of this approach, it is clear that, in the worst case scenario the number of lookups required:

$$\begin{aligned} \ell &= \ell_{insertion} + \ell_{deletion} + \ell_{substitution} + \ell_{transposition} \\ &= n + 25(n+1) + 25n + n - 1 \\ &= 52n + 24. \end{aligned}$$

This would mean that for the average length string in the LOLITA dictionary, 8 characters, a total of 440 dictionary accesses would be required. The system was found to be prohibitively slow when implemented. The general principle is preserved in the algorithms described in this chapter, however the use of the trie dictionary structure can be used to considerably reduce the amount of work required (see Section 5.5).

- Rule-Based Approach - The problem with the use of a rule-based approach in the LOLITA system is that it is language specific. LOLITA is designed to be as flexible as possible. The use of rules specific to one language should be kept to a minimum. Clearly the parser needs to be language specific, however

it is not acceptable that every time a new language is added research must be undertaken to discover spelling patterns in that language.

- Similarity Key Approach – The use of a system such as Pollock and Zamora's SPEEDCOP [POL 84] was considered, but rejected on practical grounds. Such an approach relies on the retrieval of similar words through use of the keys, then computing the edit distance between these words and the target. LOLITA currently makes use of the semantic network data structure and up to three dictionaries, depending on how many languages are being used. This places a strain on resources, both disk and internal RAM. The addition of an extra dictionary for each language was seen to be an unacceptable overhead. However, data collected in the SPEEDCOP research was used to help order the search described in this chapter.

- Probabilistic Approach – Some aspects of the probabilistic approach have been incorporated into the spelling correction system developed for LOLITA. The probabilistic approach uses the probabilities of certain types of errors occuring in order to try to correct them. The system which LOLITA uses makes use of the probabilities of the errors in order to give the search an ordering, so that the list of corrections can be generated in order of likelihood. As will be seen, these probabilities are based upon a number of factors: frequencies of letters, phonetics, keyboard layout, etc.

The overall strategy that is used in LOLITA could be described as being a reverse minimum edit distance technique, utilising probabilities in order to order the search, and a trie dictionary to cut the search space by exploiting the structure and so minimise 'garden path' searching. In a wider sense, the way in which LOLITA uses the list of spelling candidates and their associated penalties (see Section 5.4) should lead to a context sensitive choice of correction.

The dictionary storage method that has been designed for the LOLITA system has been discussed in detail in Chapter 4. The diagrams presented in this chapter, which pertain to dictionary search strategies, use a simplified version of the trie

structure to aid comprehension. Each node in the dictionary contains a symbol, a '$\surd$' represents that the string to reach this node is a valid word (equivalent to the list of `Noderefs` at that node containing entries) and a '$\times$' to represent that no word is found. It is assumed that the list format is used throughout, *i.e.* if a letter is unreachable, given the preceeding letters, no arc will be shown.

5.2 The four errors

In this section the four basic error types are discussed in greater detail and the strategy for attempting to correct each of these errors in isolation is presented. In Section 5.3 the overall spelling correction algorithm is described.

5.2.1 Insertion Errors

The removal of one character from the input string requires n lookups in the dictionary for a string of n characters. This is not a significant amount of dictionary accesses, when compared to the omission or substitution error forms.

5.2.2 Omission Errors

As part of the SPEEDCOP research, which was summarised earlier in Section 3.2.1, Pollock and Zamora [POL 84] discovered that consonants were missed from words in the following frequency order:
RSTNLCHDPGMFBYWVZXQKJ
i.e. R was omitted more often than any other letter, J the least. Using this data it is possible to order the search so that the most often omitted letter is inserted first, the least common last. The vowels will be tried before any consonants are tried, to deal with the phenomena of vowel pairs, in which usually only one vowel is sounded, (for example the pair *ou* in double).

5.2.3 Substitution Errors

Substitution errors involve the replacement of the correct letter with another. The reasons why these mistakes are made in the first place can be used to draw up a list of letters in order of likelihood of substitution for any given letter. There are two main causes of substitution errors, typing errors and phonetic confusion.

Typing Errors

As has been stated earlier, the assumption is being made that the primary input method is the qwerty keyboard as shown in Figure 1.2. The way in which people type has a pronounced effect on the pattern of errors which are found to occur in the input strings. It is these patterns which are observed to occur which influence the order in which the spelling corrector will attempt to recreate the intended input string. Two distinct typing styles are considered to be of interest.

- One handed typists – This is generally seen to be the behaviour of users who are not experienced typists. They use one hand, or often one finger, to cover the entire keyboard.
- Two handed (touch) typists – This is the expert method of typing. Each finger is assigned a column of keys to cover, as shown in Figure 5.1. The typist will then, ideally, use only the finger which controls that column for any letter in the column.

The way in which a typist uses their keyboard is of particular interest when looking at the problem of substitution errors. The spelling correction algorithm needs to try out other letters in a position currently occupied by a letter. The ordering of the list of letters to try at that position can be improved by using the information on the likelihood of other keys being hit by mistake.

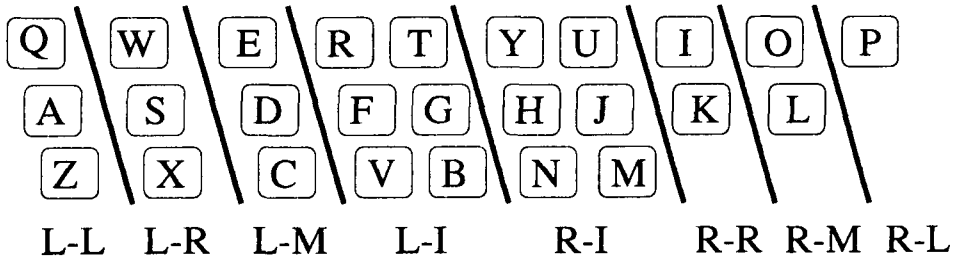


Figure 5.1: Touch typists key control

Take, for example, the letter *d*: this key is in the column controlled by the left-middle finger, so it would be expected that a larger than average number of substitutions featuring the letter *d* would be made when intending one of the other letters in this column, i.e. *e* and *c*. A typist using the one-handed approach would be expected to have hit a surrounding key by mistake. The keys which feature *d* in their group of neighbouring keys are *e*, *r*, *f*, *c*, *x* and *s*. These keys are thought to be likely candidates as the intended letter.

Phonetic confusion

The way in which a word is sounded offers a great help to spelling, however it can also be deceptive. As was demonstrated in Section 1.2.1 there is not a direct mapping between phonemes (sounds) and letters, the former outnumbering the latter 3:1. This a difficult problem faced by modern attempts at speech recognition systems. The mappings do provide the designer of spelling correction algorithms with a valuable heuristic with which to order the search strategy which is invoked. The phonetics should pick up some sound/letter ambiguity where the typist has spelt by ear, for instance spelling cake as kake.

In order to classify the likelihood of confusion between letters, due to phonetic similarity, the letters have been categorised into six groupings, approximately based on phonetic classes.

- Plosive - p, b, t, d, k, c, g, q, x, j
- Strong fricative - s, z
- Weak fricative - f, v, h
- Liquid/Glide - l, r, w, y
- Nasal - n, m
- Vowel - a, e, i, o, u

Substitution Candidates

Using the three types of substitution errors defined above it is possible to assign three groups of potential substitution letters for each of the 26 letters, for example with the letter, *d*:
1-finger = [x, s, e, r, f, c]
10-finger = [e, c]
phonetic = [p, b, t, k, c, g, q, x, j]

It is then a simple task to combine these three sets and construct an ordering with which to conduct the search, for example the letter *d* shown above becomes:
1-finger U 10-finger U phonetic = [c]
1-finger U 10-finger = [e]
1-finger U phonetic = [x]
10-finger U phonetic = []
1-finger = [s, r, f]
10-finger = []
phonetic = [t, p, g, b, k, q, j]
others = [a, i, o, n, h, l, u, m, y, w, v, z]

If there is a group containing more then one letter the list is ordered according to the letters' relative frequency in English texts, i.e.,
ETAIONSRHDLUCMFYWPGBVKQJXZ

These lists of letters allow the search to be better ordered by looking for the most likely substitutions first. Ordering the search in this way allows the use of the list of spelling corrections in a lazy manner, as will be shown in Section 5.5.

5.2.4 Transposition

The testing for transposition errors is a relatively cheap exercise, involving $n-1$ possible strings for an n character string. One heuristic which may be of use is the relationship between letters and the hand which controls them. As can be seen in Figure 5.1, half of the keys are controlled by one hand, half by the other, if two letters which are together in the string are on opposite halves of the keyboard then they would be more likely to be transposed than letters from the same group. However given that the cost of examining the $n-1$ possible strings is low, in comparison to the cost of omission and substitution error correction, the use of heuristics is felt to be unnecessary.

5.3 The overall algorithm

The algorithms described above are used to produce ordered lists of candidates. It is now necessary to combine these algorithms into one overall spelling correction algorithm. The overall algorithm must maintain the ordering produced by the four sub-algorithms, by merging their lists in a sensible manner. Given an input string the algorithms must be applied in order of likelihood, *i.e.*, the most common of the single errors must be tried first. Pollock and Zamora carried out experiments to find the order of precedence of the simple error types [POL 83]. Their survey of 50,000 errant words, found the following ordering of errors:

1. Transposition and Omission

2. Insertion

3. Substitution

The ordering of these errors is not absolute. For example, the most common insertion error would be expected to occur more often than the least common omission error. The overall search is designed to preserve the general ordering as given above, while taking these other factors into consideration. This is achieved by splitting the two errors which produce the large lists of hypothesised strings into bands, in order to allow the 'merging' of the search. The omission errors are split into five groups, the first group containing the five most often omitted letters, and so on to the final group containing the six least. The substitution errors are split in a similar way, into four groups of six. The overall algorithm takes the following form:

```
test for Transpositions
test for Band One Omissions
test for Band Two Omissions
test for Insertions
test for Group One Substitutions
test for Band Three Omissions
test for Group Two Substitutions
test for Band Four Omissions
test for Group Three Substitutions
test for Band Five Omissions
test for Group Four Substitutions
```

The spelling correction system needs to deal with more than just the simple error forms. Currently two errors per word are allowed in the system, however this figure can be easily changed if required. The strategy used is to apply the single error algorithm to a string which has already had a single error correction

applied to it. This will, inevitably, produce a much larger list of possible corrections than would single error correction. The property of laziness, described in Section 2.2.1, means, however, that these double corrections can be included in our list of theorised errors, but the work to evaluate them need only be done if they are required by the parser.

5.4 The Penalty System

LOLITA utilises a system of penalties which are accumulated by a sentence as it is parsed, a sentence which parses perfectly will accumulate no penalties. These penalty scores can then be used in order to choose between alternate parses of a sentence. The spelling correction system contributes to this scheme by allocating each word, which it returns as a possible correction, a score based on how common the error which had to be corrected to reach the word is. A word which was corrected by the transposition algorithm, for example, would be given a small penalty, a word which required a double error correction would be given a large penalty. Penalties are given in the following groupings, the actual values of the penalty are given in parentheses:

- No penalty (0) – The word itself.
- Tiny penalty (1) – Transpositions, band one and two omissions.
- Small penalty (2) – Insertions, group one substitutions and band three omissions.
- Medium penalty (5) – Group two substitutions and band four omissions.
- Normal penalty (10) – Group three and four substitutions and band five omissions.
- Big penalty (15) – Multiple errors.

These values have been set by hand, however over time it may be necessary to revise the groupings in order to maximise the effectiveness of the penalty system. The automatic optimisation of the dialogue section of LOLITA was achieved using adaptive algorithms [NET 94], and this approach may well be suitable in this area of the system also.

5.5 Implementation

The spelling correction system makes use of the trie structure described in Chapter 4. The dictionary itself is coded as an Abstract Data Type (ADT) (see Section 2.2.1) and so it is necessary that the spelling correction system be implemented in the same module in order that the internal structure of the trie may be exploited. This section documents the implementation of the four simple errors' correction algorithms, and then the overall algorithm. Finally, a discussion of the operation of the spelling correction system in conjunction with the rest of LOLITA is given.

5.5.1 Insertion Correction

An example of the search followed by the insertion error correction algorithm is given in Figure 5.2.

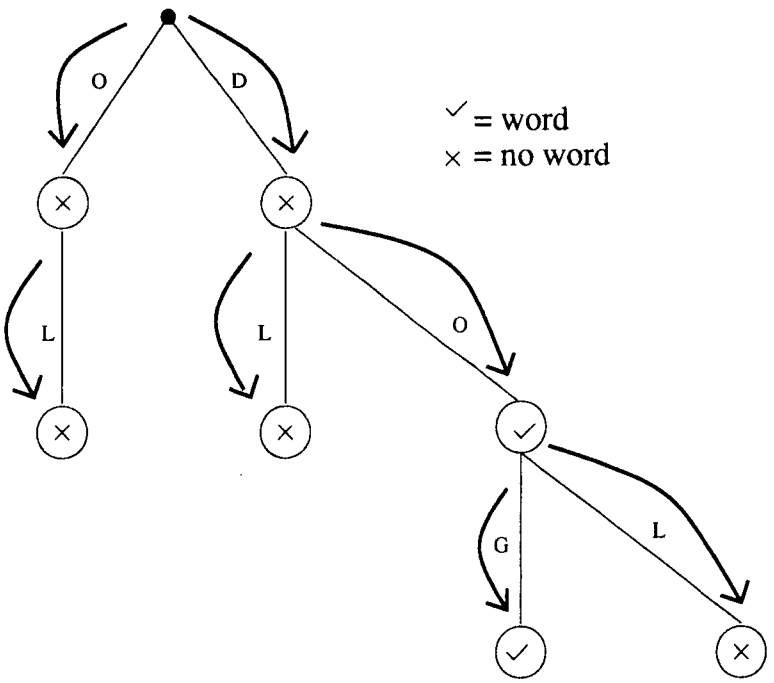


Figure 5.2: Search for insertion errors from string 'dolg'.

The search proceeds down the path which represents the input string. At each node on this path the search branches down the path which uses the next but one letter, therefore ignoring the potentially extra character. The search then continues until either all of the letters are consumed, at which point a check is made to see if a word has been found, or there is no further path available. When either a word is found, or the end of a path is reached, the search returns to the point at which the branch was made and continues down the next unexplored path.

In the example shown in Figure 5.2, the string 'dolg' is to be corrected. The search first follows the path for the string 'olg', when the path ends because there is no branch for 'g' following 'ol' the search returns to the top node and looks down the path 'olg' from 'd', again the search fails, there is no path for 'g' to follow 'dl'. The search does, however, find a word on its next attempt, branching at the node

which represents words beginning 'do' the search branches down the 'g' arc, using the last of its letters. Since the node represents a valid word, 'dog', the word is added to the list of spelling candidates and the search resumes. The search ends at the node for 'dol', having consumed all letters, but not at a valid word. The insertion correction is now complete and the search moves on to looking for other types of errors.

The example shows one way in which the trie structure aids the search. When the search down the path 'dl' fails the algorithm only backtracks as far as the last branch in order to resume searching. This way the number of arcs traversed between nodes is considerably reduced. Were the candidates generated, and then tested, ten arcs would have been traversed instead of seven (the 'd' arc three times and the 'o' arc twice). With larger words the saving is considerable.

5.5.2 Omission Correction

Omission errors are corrected through the insertion of a series of letters in-between the characters in the string. The letters to be inserted come from the list introduced in Section 5.2.2. An example of the way in which this algorithm operates on the trie is shown in Figure 5.3.

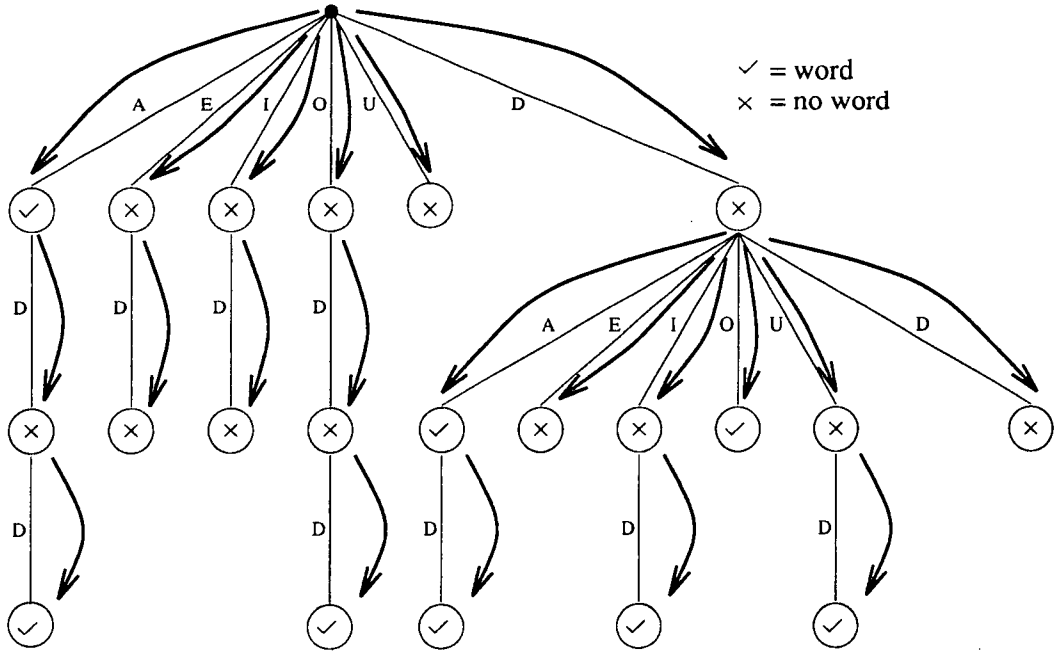


Figure 5.3: Search for group one omission errors from string 'dd'.

The figure shows the way in which the search proceeds through the trie. At the top node the search proceeds down the first of the candidate strings paths, eventually finding the word 'add' which is added the list of hypothesised spelling corrections. The search then returns to the last node with unexplored paths, in this case the top node, and tries the next unexplored path. After completing the search for the omission of the first letter of the word, the search progresses attempting to insert a character after the first letter. In this case the 'a' ('dad'), the 'i' ('did') and the 'u' ('dud') all produce valid words which are appended to the list of candidates. The search now looks for letters added to the end of the word. As there are no paths from 'dd' the search will end here and pass control back to the overall search controller.

5.5.3 Transposition Correction

The correction of transposition errors is computationally the cheapest of all the algorithms. The search involves following the path for the given string and at each node trying the next but one letter in the string ahead of the next character as shown in Figure 5.4.

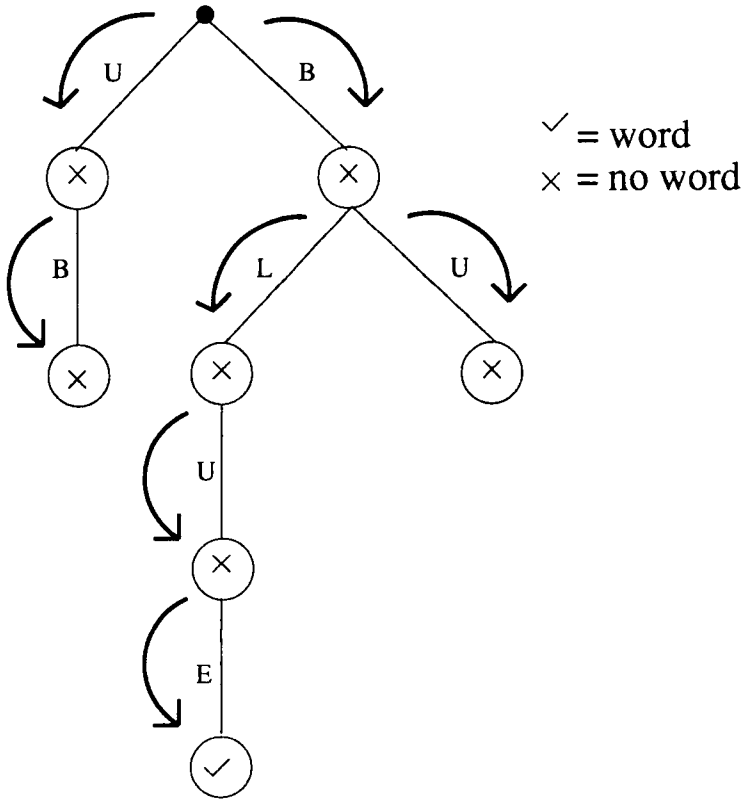


Figure 5.4: Search for transposition errors from string 'bule'.

A four character string, as shown in the diagram, only requires the search to look down three different paths. In the example shown, the search first looks for the transposition of the first two characters, resulting in the string 'uble', this path fails as there is no subtrie for an 'l' to follow 'ub'. Next the second pair of characters are swapped, resulting in the word 'blue' which would be added to the list of corrections. The algorithm then returns to the point at which the last deviation from the string itself was made and resumes the search which again fails, as there is no words starting 'bue'. The transposition error correction is now complete.

5.5.4 Substitution Correction

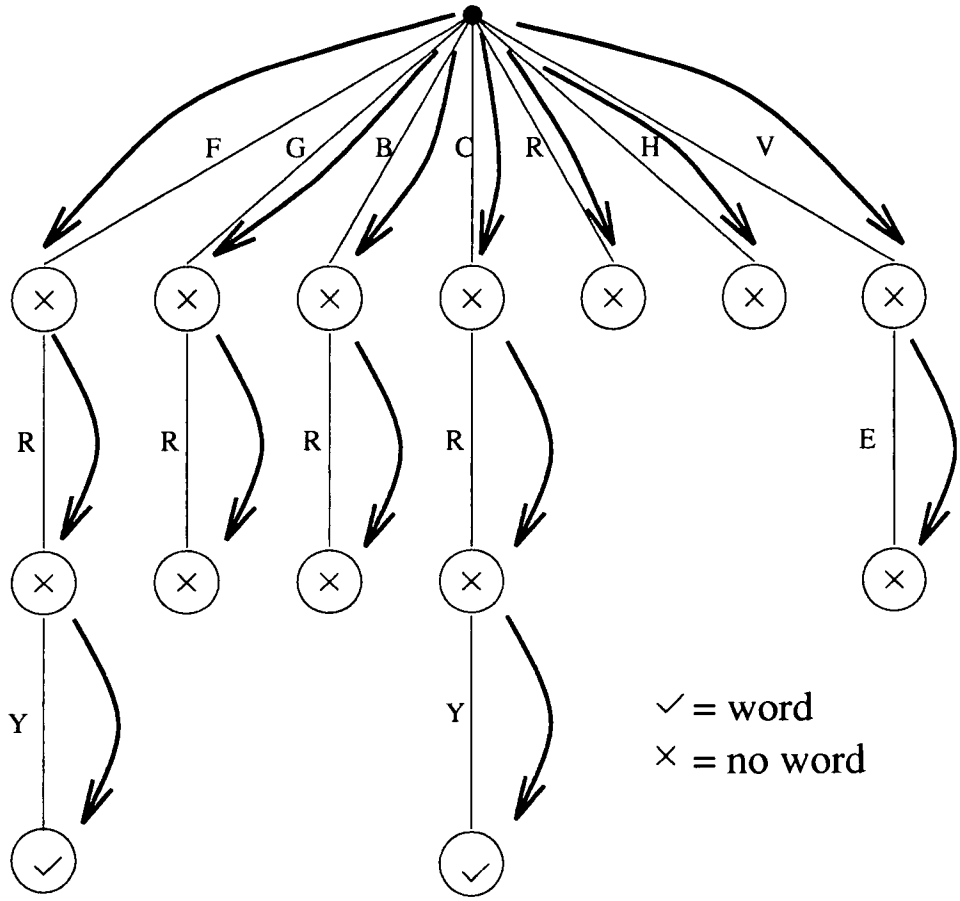


Figure 5.5: Search for band one substitution errors from string 'vry'.

As was discussed in Section 5.2.3 the substitution algorithm makes use of an ordered list of candidate letters, for each letter found in the string. The algorithm to look for a substitution error is demonstrated in Figure 5.5. The string 'vry' is subjected to the correction of band one substitution errors. The letter 'v' has as its band one substitution list the letters 'fgbcrh'. These letters are tried in turn in place of the letter 'v'. In the first attempt, the letter 'f' produces the word 'fry', this word is added to the list of possible corrections, and the next substitution tried. The letters 'g' and 'b' both fail to produce a valid word. The word 'cry' is formed and added to the list, the final two letters fail to get past the one letter stage in their attempts to build a word. The search then continues, this time accepting the letter 'v' as the words first character and trying substitutions on the second letter in the word, 'r'. The only one of the substitutions for 'r' that can follow 'v' as the second

letter of a word is 'e' but this can not be followed by the letter 'y'. The final letter of the word is not considered for substitution as no word could start 'vr' in order to get to the point in the trie where such words would be located.

This search tree is a very good example of the way in which the trie structure is of considerable aid to cutting the search space. A standard implementation of the reverse minimum edit distance correction technique would require the lookup of eighteen words (allowing only six substitutions per character). Six of the lookups would have been for words which started with the impossible (starting) string 'vr'. It is this type of wasted lookup which the trie implementation avoids.

5.5.5 Overall Algorithm

The four algorithms described above are combined into the unified search strategy described in Section 5.3. The function `corrections` produces a list of pairs of words and penalties, for example `("fish", smallpen)`. It is important, in order to save the parser doing the same work more than once, that the words produced in this list are unique. The problem could arise in the following way: consider the error string 'spelll', the word would be corrected to the same word, 'spell' by correcting an insertion error in the 4th, 5th and 6th positions. This problem is especially important in the correction of a double error, in a real-word. Every string with one correction attempt, which is then passed for its second, can have the first correction corrected back to produce the original word. This problem is overcome by filtering the list so that it only produces one pair containing that word. The spelling correction will produce, in a lazy manner, a list of the form shown in Figure 5.6.

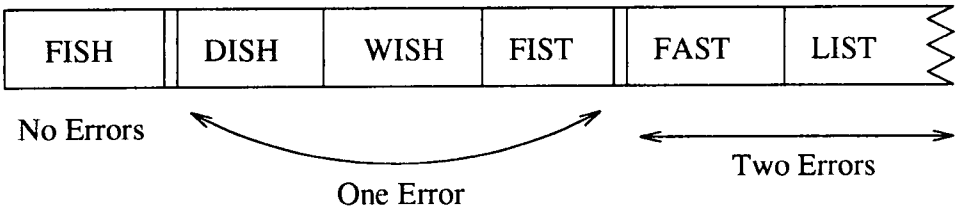


Figure 5.6: A list of corrections for the word 'fish'.

The list contains the word itself, if it is valid, the single error corrections, and finally the multiple error corrections. The word itself is included at the head of the list as it is intended that, rather than calling the spelling correction system only if an error is detected, each word in the sentence being parsed will consist of a list of possible words. It is the lazy method of evaluation, described in Section 2.2.1, which makes this approach practical.

The construction of the list of multiple errors is done using the function 'rest-word'. The function is responsible for retrieving the rest of a word once an error has been corrected. The function restword, however, is designed to find the rest of the word only if the specified number of corrections has been performed on the word, otherwise the rest of the word is passed to the spelling corrector in order to introduce a further correction.

5.6 Integration with LOLITA

There are a number of issues to be addressed in the integration of the spelling correction system with LOLITA. These issues involve both the way in which the system interacts with the rest of LOLITA as it currently stands, as well as the way in which the increased functionality provided by the spelling corrector impacts upon future developments of certain sections of LOLITA.

- The parser – Currently the parser used in LOLITA is not designed to utilise the penalties generated by the spelling correction system. For this reason the parser does not treat the stream of spelling correction candidates in a lazy manner. This means that the parser will attempt to parse sentences containing each of the spelling candidates in turn, before sorting them into order based upon the suitability of their parsing costs. This can be demonstrated using the word 'liek'. The list of (single) spelling error candidates, in order of generation, is: like, lie, lick, leek, link, lieu. However the sentences which the parser generates, in order of its preference, are:

```
I LICK [Misspelt] ROBERTO
I LINK [Misspelt] ROBERTO
I LIEK [New] ROBERTO
I LIKE [Misspelt] ROBERTO
```

The new parser, currently under development, is being designed to incorporate the new functionality which the spelling correction system provides, and therefore will help solve these problems. As a temporary measure to limit the amount of parsing done by the system, the spelling correction system will only generate multiple error candidates if no single error candidates can be found.

- Semantic and Pragmatic selection - The semantic and pragmatic analysis modules of LOLITA are the next stage at which disambiguation takes place. Any parse trees of sufficient quality will be ordered by the pragmatics according to which makes the most sense in terms of LOLITA's world knowledge. An ongoing research project is currently investigating the concept of semantic distance. It is hoped that in future the spelling correction system will be provided with a cut down dictionary containing only words which are within a certain distance of the anticipated meaning of the word at that point in the sentence.

- Real Word Errors - A major class of errors which are impossible to detect or correct in isolation are real word errors. The two ways which these errors occur are the transformation of the intended word into another valid word (e.g. 'dish' for 'fish') and the use of a word in the wrong context (e.g. 'to', 'too' and 'two'). Now that each word in a sentence can be thought of as a list of possible words at that point in the sentence, the parser or semantic analysis components of LOLITA are able to try the next word in the list at the point at which the sentence fails. There is an important issue to be addressed by the system, however, when dealing with errors of this type. Given the sentence 'I pray to my dog', is the intended sentence 'I pray to my god' (in which case making the correction at the point of failure is correct)?

Or is it 'I play to my dog' (in which case the error was actually earlier in the sentence)? Clearly the use of even wider context is required in this case. For example, if the preceeding text read 'I like to practice the piano, although not to an audience' the latter interpretation of the sentence would be prefered.

- Word-breaks – The CLARE system described in Section 3.2.2 used a lattice data structure for input to deal with word-breaks. The input stream of characters in LOLITA is treated as a list of words, separated by either punctuation marks or spaces. The algorithms given above could be applied to a lattice-based system. Adding this functionality would simply be a case of combining the spelling correction and lattice structure. This was experimented with, briefly, and found to produce a very large number of possibilities. It was felt that, due to the current structuring of the LOLITA system, the addition of a lattice based approach was not worth the additional computations and recoding it would require. Clearly, inclusion of the word-break problem is one which merits further research.

Inevitably, the addition of new functionality in one area of the system will result in a number of additional facilities which can be exploited by the rest of the system. The features described above, when implemented, will increase the overall functionality of LOLITA over and above the benefits which were intended originally, i.e. to provide a fast solution to the spelling correction problem.

5.7 Testing

There are two issues involved in testing the spelling correction system. Firstly does the code exhibit the functionality which is expected. Secondly, does the functionality which the system provides meet the requirements of the environment in which it is to be used. The testing of the code itself was a relatively trivial task, and involved simply checking that when given a string to correct the list of words returned contains all of the words which should be present and no more. In order

	1	2	3	4	5+	none
LOLITA	93	2	2	0	2	1
Word	83	7	4	2	1	3
ispell	76	9	4	1	5	5

Table 5.1: position of word in list

to evaluate the performance of the spelling correction system, beyond the "does the code do what is should?" stage it is first necessary to establish the criteria by which it will be judged.

The system produces a list of corrections, therefore the main questions to be considered are:

1. Does the list of corrections produced contain the target word?
2. If so, where in the list does it occur?

In order to test the system a corpus of errors was obtained, which contained spelling errors printed in "The Times" newspaper. This data-set was seen as ideal for use in this task as newspaper articles are used by the contents scanning tool. From this corpus 100 words were selected, according to numbers provided by a random number generator. Each of these 100 words was given to the spelling correction system, and a note was made of at which position within the list the intended word appeared; a full list of the words, and the corrections produced, are given in the appendix. This exercise was then repeated using the UNIX *ispell* system and the spell-check system used by a popular word-processor, *Microsoft Word for Windows*. The results are given in Table 5.1.

These results were plotted onto a cumulative frequency graph, so that the number of words classified by each position can be compared. This graph, as seen in Figure 5.7, shows that the spelling correction system used by LOLITA compares favourably with the commercial systems tested as benchmarks. The main difference is the number of times that LOLITA's system finds the correct word first time. In a more general situation as long as the word is presented to the user as

a possibility the ordering is not important. However, the ordering of the search is more important in the context of LOLITA, as the amount of extra work which can be saved by getting it right first time is so great. These results show how the use of the ordering heuristics (see Section 5.2) has enabled the system to achieve impressive 'first try' statistics.

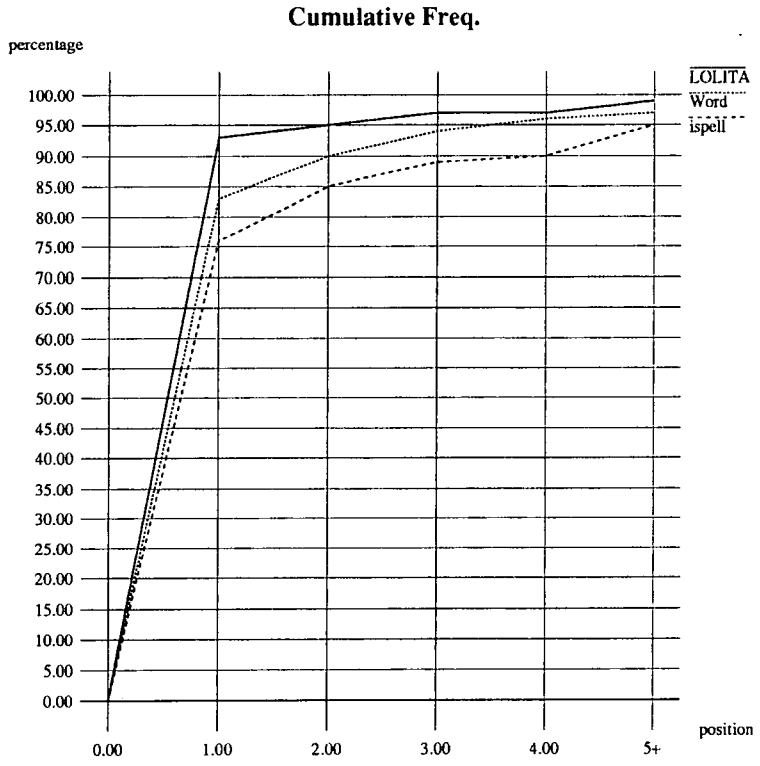


Figure 5.7: Cumulative percentages for list position

5.8 Summary

In this chapter the spelling correction system has been introduced. The system makes extensive use of the trie dictionary described in the previous chapter in order to aid the search for possible corrections. An adaptation of the 'reverse minimum edit distance' strategy is used to hypothesise corrections, using probabilities to order the search. For each of the four error types the search is ordered based on the likelihood of each of the different types of error. The overall search splits the two largest searches, (Omissions and Substitutions) into sections in order to

allow the overall search to be interleaved so that the correction list is produced in order of likelihood. The way in which the trie structure speeds up the search was shown, and this was the main factor behind the introduction of tries for the dictionary. The effects of lazy evaluation on the production of the list of spelling corrections was discussed and seen to be a most desirable feature. The integration with LOLITA raised a number of issues in relation to the functionality of other parts of LOLITA which can be addressed by future developments of the system. Finally, in testing the system a set of criteria were given in order to evaluate the usefulness of the spelling correction system, and the performance seen to be at a level which compares favourably with existing commercial systems.

Chapter 6

Conclusions

Chapter 1 introduced the problem addressed by this work and provided an overview of the issues involved in spelling correction and dictionary organisation. The LOLITA Natural Language Processing system can be used to read articles, transcribed from newspapers and analyse the content of the article, filling a template with the relevant details. In order to make the system more robust the facility to correct any misspellings in the text, without resorting to human intervention, was required.

Chapter 2 described the context within which the system was to be developed and used, and this was shown to be very influential upon the design of the system. The programming features which are provided by the functional programming language Haskell were of particular importance. The use of lazy evaluation, in which an expression is only evaluated if its result is actually required, was shown to be an extremely useful feature when seamless dictionary partitioning and hypothesised correction generation were considered. The use of Abstract Data Types (ADTs), coupled with the property of purity possessed by Haskell, made the integration of the new data-structures and algorithms into the existing system a relatively easy task.

Various different strategies for both spelling correction and dictionary organi-

sation were compared and contrasted in Chapter 3.

The development of a dictionary structure based upon the use of an M-ary tree (trie) system was presented in Chapter 4. The use of tries was seen to be influenced by two main factors, the independence of dictionary access time from dictionary size, and the help which the structure would offer to the spelling correction system. The implementation in Haskell of the dictionary was seen to make use of either arrays of lists at each node, depending upon the number of nodes, in order to minimise the access time for a word. A number of implementation features were described, in particular the way in which lazy evaluation assisted in the partitioning and loading of the dictionary. The technique of profiling was introduced as a way of 'tuning' the data-structure for optimal performance. The dictionary has proved to be flexible enough to cope with the new data set, now used by LOLITA, which contains almost twice as many words as the data-set in use during development.

Finally, Chapter 5 introduced the spelling correction algorithms which make use of the dictionary described above. The four simple error forms (insertion, omission, transposition and substitution) were examined in turn and strategies developed to generate corrections for each. These strategies use various heuristics in an attempt to generate the list of corrections in order of likelihood. Furthermore, the two most expensive searches, those for omissions and substitutions, were split into smaller sub-searches, so that the overall algorithm can generate them in order too. The dictionary design was seen to greatly assist the search by limiting the amount of work which needed repeating, and by preventing 'garden path' searching. A number of issues were raised in the discussion of integration with LOLITA, primarily focusing on the issue of using syntactic or semantic information in order to disambiguate alternative hypotheses. Criteria by which the performance of the spelling correction system may be judged were given, and the system seen to compare favourably with other systems widely available.

Bibliography

[ALS 84] Al-Suwaiyel M. and E. Horowitz, "Algorithms for Trie Compaction," ACM Trans. on Database Systems, Vol. 9., No. 2., pp 243–263, 1984.

[ANG 83] Angell R. C., G. E. Freund and P. Willett, "Automatic spelling correction using a trigram similarity measure," Inf. Process. Manage., Vol. 19, No. 4, pp 255–261, 1983.

[AOE 92] Aeo, J.-I., K. Morimoto and T. Sato, "An Efficient Implementation of Trie Structures," Software-Practice and Experience, Vol. 22, No. 9, pp 695–721, 1992.

[CAR 92] Carter D. M., "Lattice-based word identification in CLARE," In Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics, Delaware USA, pp 159–165, June 1992.

[CHA 54] Chapanis A., "The reconstruction of abbreviated printed messages," J. Exp. Psychol, Vol 48. pp. 496–510, 1954.

[CLO 84] Clocksin W. F. and C. S. Melish, "Programming in Prolog, 2nd ed.," Springer-Verlag, New York, 1984.

[COM 81] Comer D., "Analysis of a Heuristic for Full Trie Minimization," ACM Trans. on Database Systems, Vol 6., No. 3, pp 513–537, 1981.

[CRY 87] Crystal D., "The Cambridge Encyclopedia of Language," Cambridge University Press, 1987.

[DAM 64] Damerau F. J., "A Technique for Computer Detection and Correction of Spelling Errors," Commun. ACM, Vol. 7, No. 3, pp 171–176, 1964.

[DAM 89] Damerau F. J. and E. Mays, "An examination of undetected spelling errors," Inf. Process. Manage. Vol. 25, No. 6, pp 659-644, 1989.

[DOW 82] Downton A. C., "Simultaneous Transcription of Machine Shorthand for the Deaf," University of Southampton 1982.

[DUN 81] Dunlavey M. R., "On Spelling Correction and Beyond," Commun. ACM, Vol. 24, No. 9, 1981.

[DUR 83] Durham I., D. A. Lamb and J. B. Saxe, "Spelling correction in user interfaces," Commun. ACM, Vol. 26, No. 10, pp 764–773, 1983.

[ELL 93] Ellis N. R., R. Garigliano and R. G. Morgan, "A New Transformation into Deterministically Parsable Form for Natural Language Grammars," In Proceedings of the 3rd International Workshop on Parsing Technologies, Tilburg, Netherlands, 1993.

[FRE 60] E. Fredkin, "Trie Memory," Commun. ACM, Vol. 3, No. 9, pp 490–500, 1960.

[GAR 92] Garigliano R., R. G. Morgan and M. H. Smith, "LOLITA: Progress Report 1," Technical Report, Durham University, 1992.

[GAR 93] Garigliano R., R. G. Morgan and M. H. Smith, "The LOLITA System As A Contents Scanning Tool," In Proceedings of Avingnon '93, 1993.

[DAR 91] Defense Advanced Research Projects Agency, "Proceedings of the Third Message Understanding Conference," DARPA, 1991.

[HAL 80] Hall P. A. V. and G. R. Dowling, "Approximate string matching," ACM Computing Surveys, Vol. 12, No. 4, pp 381–402, 1980.

[HAN 76] Hanson A. R., E. M. Riseman and E. Fisher, "Context in Word Recognition," Patt. Recog., Vol. 8, pp 35–45, 1976.

[HAR 72] Harmon L. D., "Automatic Recognition of Print and Script," Proc. IEEE, Vol 60, No. 10, pp 1165–1176, 1972.

[HAZ 93] Hazan J. E., S. A. Jarvis, R. G. Morgan and R. Garigliano, "Understanding LOLITA: Program Comprehension in Functional Languages," Workshop on Program Comprehension, IEEE Computer Society Press, Capri, Italy, 1993.

[HOT 80] Hotopf N., "Slips of the pen," in "Cognitive Processes in Spelling," Uta Frith, Ed., Academic Press, London, pp 287–307, 1980.

[HOL 91] Holyer I. J., "Functional Programming with Miranda," Pitman Publishing, London, 1991.

[HUD 92] Hudak P. *et. al.*, "Report on the Functional Programming Language Haskell," Version 1.2. 1992.

[JON 93] Jones C. and R. Garigliano, "Dialogue Analysis and Generation: A theory for modelling natural English dialogue," in Proceedings of EUROSPEECH '93, Vol. 2. p 951., Berlin, Germany, 1993.

[KNU 73] Knuth D. E., "The art of programming vol. 3.: searching and sorting," Addison-Wesley, Reading, Mass, 1973.

[KUK 92a] Kukich K., "Spelling Correction for the Telecommunications Network for the Deaf," Commun. ACM, Vol. 35, No. 5, pp 80–90, 1992.

[KUK 92b] Kukich K., "Techniques for Automatically Correcting Words in Text," ACM Computing Surveys, Vol. 24, No. 4, pp 377–439, 1992.

[LON 94] Long D. and R. Garigliano, "Reasoning by Analogy and Causality," Ellis Horwood, 1994.

[MOR 70] Morgan H. L., "Spelling Correction in Systems Programs," Commun. ACM, Vol. 13, No. 2, pp 90–94, 1970.

[MOR 94] Morgan R. G., R. Garigliano, S. A. Jarvis and B. S. Parker, "The LOLITA System as an Example of Large Scale Functional Programming,"

Dagstuhl Workshop on Funct. Prog. in the Real World (invited paper), Dagstuhl, Germany, May 16-20, 1994.

[MUT 77] Muth F. E. Jr. and A. L. Tharp, "Correcting human error in alphanumeric terminal input," Inf. Process. Manage., Vol. 13, pp 329–337, 1977.

[NET 94] Nettleton D. J. and R. Garigliano, "Evolutionary Algorithms for Dialogue Optimisation in the LOLITA Natural Language Processor," Seminar on Adaptive Computing and Information Processing, Jan '94, 1994.

[PET 80] Peterson J. L., "Computer Programs for Detecting and Correcting Spelling Errors," Commun. ACM, Vol. 23, No. 12, pp 676–687, 1980.

[PET 86] Peterson J. L., "A note on undetected spelling errors," Commun. ACM, Vol. 29, No. 7, pp 633–637, 1986.

[POL 82] Pollock J. J., "Spelling Error detection and correction by computer - some notes and a bibliography," J. Doc., Vol. 38, No. 4, pp 282–291, 1982.

[POL 83] Pollock J. J. and A. Zamora, "Collection and characterization of spelling errors in scientific and scholarly Text," J. Am. Soc. Inf. Sci. Vol. 34, No. 1, pp 51–58, 1983.

[POL 84] Pollock J. J. and A. Zamora, "Automatic Spelling Correction in Scientific and Scholarly Text," Commun. ACM, Vol. 27, No. 4, pp 358–368, 1984.

[POT 91] Potter B., J. Sinclair and D. Till, "An introduction to Formal Specification and Z," Prentice-Hall International, 1991.

[RAM 89] Ramesh R., A. J. G. Babu and J. P. Kincaid, "Variable-Depth Trie Index Optimization: Theory and Experimental Results," ACM Trans. on Database Systems, Vol. 14., No. 1., 1989.

[SAM 94] Sansom P. M., "Execution Profiling for Non-strict Functional Languages," Ph.D. Thesis, University of Glasgow, 1994.

[SEB 93] Sebesta R. W., "Concepts of Programming Languages, 2nd Ed.," Benjamin/Cummings Publishing Co. 1993.

[SEL 69] Selinker L., "Language Transfer," General Linguistics 9, pp69–92, 1969.

[SHA 88] Shastri L., "Semantic Networks: An evidential Formalization and its Connectionist Realisation," Morgan Kaufmann, 1988.

[SMI 94] Smith M. H., R. Garigliano and R. G. Morgan, "Generation in the LOLITA system: An engineering approach," Seventh International Workshop on Natural Language Generation, Maine, USA, 1994.

[ULL 77] Ullmann, J. R., "A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words," Comput. J., Vol. 20, No. 2, pp 141–147, 1977.

[WAN 92] Wang Y. and R. Garigliano, "An Intelligent Tutoring System for Handling Errors Caused by Transfer," in Lecture Notes in Computer Science 608: Proceedings of Second International Conference on Intelligent Tutoring Systems, pp. 395–404, Springer-Verlag, Montreal, Canada, 1992.

[YAN 83a] Yannakaoudakis E. J. and D. Fawthrop, "The Rules of Spelling Errors," Inf. Process. Manage., Vol. 19, No. 2, pp 87–99, 1983.

[YAN 83b] Yannakoudakis E. J. and D. Fawthrop, "An Intelligent Spelling Error Corrector," Inf. Process. Manage., Vol. 19, No. 2, pp 101–108, 1983.

Appendix A

This appendix gives the full results of the testing on the LOLITA spelling correction system. Firstly the misspelt word is given, followed by the list of corrections given by LOLITA, in the order in which they are generated.

```
"accesory" ["accessory", "accessary", "ancestry"]
"accomodate" ["accommodate"]
"actuallly" ["actually", "actuality"]
"adminstrator" ["administrator"]
"advanatgeous" ["advantageous"]
"alchol" ["alcohol", "archil", "anchor"]
"almos" ["alms", "salmon", "almost", "almond", "amok", "also", "aloes",
         "aloe", "alma", "arms", "armor", "alias", "aloof", "adios",
         "allis", "allow", "allot", "alloy"]
"analagous" ["analogous"]
"aniversary" ["anniversary", "adversary"]
"appliction" ["application", "affliction"]
"archibishop" ["archbishop"]
"assocation" ["association", "avocation", "allocation"]
"attact" ["attract", "attack", "attach", "tact", "tract", "attar",
          "intact", "attest", "attain", "attache"]
"bacause" ["because", "cause", "accuse"]
"balace" ["balance", "palace", "bullace", "ballade", "lace", "place",
          "brace", "blade", "blare", "blame", "blaze", "black", "bale",
          "valance", "belate", "ablate", "palate", "solace", "malice",
          "ablaze"]
"becasue" ["because", "became"]
"beleive" ["believe", "beehive", "belie", "belove", "bereave", "deceive",
           "relieve", "receive"]
"betweeen" ["between"]
"broacaster" ["broadcaster"]
"childen" ["children", "holden", "hidden", "chicken"]
"coalitionl" ["coalition"]
"commision" ["commission", "concision", "commotion", "communion",
             "collision"]
"committment" ["commitment"]
"comprehesive" ["comprehensive", "compressive"]
"compresssion" ["compression"]
```

```
"concesssion" ["concession", "concussion", "confession"]
"cricitise" []
"dicate" ["dictate", "dilate", "abdicate", "educate", "dedicate",
          "medicate", "indicate", "delicate", "irate", "date",
          "dice", "dicta", "fixate", "donate", "debate", "decade",
          "derate", "ideate", "pirate", "dilute", "locate", "vocate",
          "vacate", "dickie"]
"diffiicult" ["difficult"]
"dimissal" ["dismissal"]
"disretion" ["discretion", "disruption", "desertion"]
"eary" ["aery", "early", "wary", "earn", "nary", "earl", "weary", "easy",
        "vary", "ovary", "unary", "ready", "dreary", "scary", "seamy",
        "terry", "tears", "nearby", "leery", "leafy", "learn", "bleary",
        "leaky", "entry", "earth", "earthy" "ray", "airy", "arty", "wry",
        "art", "arm", "try", "are", "awry", "army", "dry", "cry", "fry",
        "any", "arc", "pry", "amy", "arb", "ark", "erg", "ern", "err",
        "ere", "era", "eat", "ear", "dray", "diary", "dairy", "day",
        "dart", "darn", "dory", "dare", "darky", "dark", "saury", "say",
        "sari", "sard", "rare", "racy", "warty", "way", "wiry", "wart",
        "warn", "warm", "wavy", "ware", "ward", "waxy", "warp", "carry",
        "cay", "cart", "cavy", "care", "card", "carp", "emery", "every",
        "eats", "hearty", "heady", "heart", "henry", "heavy", "heard",
        "decry", "pearly", "peavy", "pearl", "peaky", "gears", "eagre",
        "tray", "tarry", "tart", "tardy", "tory", "tare", "tara", "taro",
        "tarp", "nay", "navy", "nard", "narc", "hairy", "harry", "hay",
        "hart", "harm", "hardy", "harpy", "hare", "hard", "hazy", "harp",
        "hark", "envy", "eave", "meaty", "merry", "mealy", "faery",
        "ferry", "beady", "berry", "beany", "beard", "yearn", "years",
        "lay", "lady", "lory", "lacy", "lard", "lazy", "lark", "marry",
        "may", "mart", "marl", "mare", "mara", "many", "marc", "mark",
        "fray", "fairy", "fay", "fart", "farm", "fury", "fare", "faro",
        "yarn", "yard", "pray", "parry", "party", "pay", "part", "pare",
        "para", "park", "gray", "gay", "gory", "gari", "garb", "eddy",
        "edgy", "ecru", "eyra", "east", "ease", "each", "very", "veery",
        "kerry", "jerry", "bray", "bay", "baby", "bart", "barn", "barm",
        "bury", "bare", "bars", "bard", "barb", "bark", "vara", "kaury",
        "jay", "jury"]
 "elswehere" ["elsewhere"]
"emnity" ["enmity", "amenity", "unity", "entity", "empty", "sanity",
          "amity", "vanity", "equity"]
"encylopaedia" ["encyclopaedia"]
"enginering" ["engineering"]
"enviroment" ["environment"]
"equaltiy" ["equality", "quality"]
"exept" ["exempt", "exert", "except", "extent", "sept", "wept", "lept",
         "kept", "exit", "swept", "slept", "crept", "adept", "exalt",
         "exact", "expect", "expert", "inept", "eject", "exist", "exult",
         "erect", "erupt", "elect", "event"]
"familar" ["familiar", "familial", "family", "basilar", "similar"]
"fimal" ["final", "simal", "animal", "formal", "fiscal", "finial",
         "filial", "imam", "foal", "fail", "fiat", "fill", "vial",
         "vital", "tidal", "tical", "rial", "primal", "riyal", "rival",
         "gimel", "gimbal", "dismal", "dial", "focal", "fugal", "fetal",
         "fatal", "sial", "somal", "sisal", "pipal"]
"fixure" ["fixture", "figure", "failure", "flexure", "fissure", "inure",
          "fire", "fibre", "future", "fixate", "mixture"]
"foresaking" ["forsaking"]
```

```
"galant" ["gallant", "sealant", "garland", "slant", "plant", "grant",
          "glint", "giant", "gland", "gaunt", "galax", "talent",
          "valiant", "valent", "vacant", "galena", "galoot", "savant",
          "galago", "galaxy"]
"histgory" ["history"]
"inagural"  ["inaugural", "natural", "binaural"]
"infinte" ["infinite", "infinity", "infringe", "invite", "incite",
           "inflate"]
"injuction" ["injunction", "induction", "injection", "inunction",
             "inaction", "intuition", "indiction", "infection"]
"insistance" ["insistence", "instance", "assistance", "insistency",
              "resistance"]
"inspetor" ["inspector", "inceptor"]
"introdusec" ["introduce"]
"irreponsible" ["irresponsible"]
"isssue" ["issue", "tissue"]
"laater" ["later", "laager", "latter", "elater", "slater", "water",
          "eater", "aster", "tater", "rater", "hater", "dater", "cater",
          "mater", "alter", "after", "pater", "bater", "liter", "lager",
          "layer", "laver", "lather", "laser", "latex", "patter",
          "raster", "ratter", "rafter", "waster", "waiter", "leaver",
          "leather", "letter", "leader", "litter", "lifter", "lawyer",
          "plaster", "platter", "planter", "easter", "taster", "tatter",
          "neater", "seater", "stater", "skater", "loiter", "loafer",
          "looter", "loader", "loaner", "luster", "lauder", "larder",
          "floater", "bloater", "heater", "hatter", "halter", "darter",
          "caster", "crater", "master", "matter", "falter", "ladder",
          "grater", "garter", "beater", "boater", "batter", "banter",
          "barter"]
"leasdership" ["leadership", "readership"]
"lieutentant" ["lieutenant"]
"likly" ["likely", "lily", "lilt", "like", "oily", "rilly", "wily",
         "lolly", "limey", "silky", "silly", "liken", "hilly", "milky",
         "filly", "lively", "lindy", "lisle", "billy", "bialy"]
"mangement" ["management"]
"massionary" ["missionary"]
"neccessary" ["necessary", "accessary"]
"othr" ["other", "rotor", "otter", "ottar", "hr", "the", "tar", "thy",
        "tor", "or", "our", "oar", "ohm", "ether", "athar", "atar",
        "over", "tour", "pother", "star", "stir", "odor", "ocher",
        "mother", "motor", "bother", "torr", "voter"]
"paramilatary" ["paramilitary"]
"pas" ["pass", "as", "das", "gas", "paw", "pax", "pad", "pus", "pat",
       "pan", "par", "has", "pal", "pay", "was", "vas", "pap", "apar",
       "epos", "eyas", "opus", "opah", "opal", "patas", "peat", "pear",
       "pecs", "peal", "peak", "piss", "puss", "pains", "paid", "pain",
       "pair", "pail", "spats", "spat", "span", "spar", "spay", "taps",
       "prat", "pram", "pray", "plans", "plus", "plat", "plan", "play",
       "parts", "pare", "para", "part", "park", "past", "pants", "pate",
       "path", "pane", "pant", "pang", "pale", "pall", "palm", "sa",
       "ass", "s", "is", "ax", "ad", "os", "us", "at", "ai", "an", "ah",
       "ms", "am", "pe", "pa", "pi", "oats", "oat", "oar", "oaf",
       "oak", "lass", "law", "lax", "lad", "lat", "lac", "lam", "lay",
       "lap", "lag", "lab", "taw", "tax", "tad", "tat", "tao", "tan",
       "tar", "tau", "tam", "tap", "tag", "tab", "dais", "dad", "dah",
       "dal", "dam", "day", "dag", "dab", "caw", "cad", "cos", "cat",
       "can", "car", "ccs", "cam", "cay", "cps", "cap", "cab", "gad",
```

```
        "gat", "gar", "gal", "gam", "gay", "gap", "gag", "gab", "psi",
        "pew", "pee", "pea", "pet", "pen", "per", "pep", "peg", "pie",
        "pia", "pit", "pin", "pip", "pig", "ape", "page", "pave", "papa",
        "bias", "bars", "bass", "baa", "bad", "bus", "bat", "ban", "bar",
        "bah", "bam", "bay", "bag", "kos", "jaw", "jar", "jaws", "jam",
        "jay", "jag", "jab", "jak", "eats", "eat", "ear", "pow", "pod",
        "pot", "poi", "pol", "pop", "put", "pun", "pul", "pup", "pug",
        "pub", "pro", "pry", "phi", "apt", "pawn", "pawl", "its", "nay",
        "nap", "nag", "nab", "sass", "sans", "sis", "saw", "sax", "sad",
        "sat", "sac", "say", "sap", "sag", "raw", "rad", "rat", "ran",
        "ram", "ray", "rap", "rag", "raj", "his", "haw", "had", "hat",
        "ham", "hay", "hap", "hag", "haj", "pdl", "ply", "pya", "mass",
        "maw", "mad","mrs", "mat", "mao", "man", "mar", "mac", "may",
        "map", "mag", "fax", "fad", "fat", "fan", "far", "fay", "fag",
        "yes", "yaw", "yah", "yaws", "yam", "yap", "yak", "waw", "wax",
        "wad", "wads", "wan", "war", "wah", "ways", "way", "wag", "vis",
        "vat", "van", "var", "val", "zap", "zag", "ppm"]
"penguiin" ["penguin"]
"petititon" ["petition"]
"pf" (of) ["f", "of", "pe", "pa", "pi", "if", "apt", "ape", "elf", "emf",
      "opt", "oaf", "off", "pat", "par", "pad", "pan", "pal", "pay",
      "paw", "pap", "pax", "pelf", "pet", "per", "peg", "pee", "pea",
      "pen", "pew", "pep", "pit", "pig", "pie", "pia", "pin", "pip",
      "pouf", "pot", "pod", "poi", "pol", "pow", "pop", "put", "pug",
      "pub", "puff", "pun", "pus", "pul", "pup", "ref", "rpm", "spa",
      "spy", "pro", "pry", "psi", "ply", "v", "t", "r", "g", "d",
      "c", "b", "h", "e", "a", "i", "o", "n", "s", "l", "u", "m", "y",
      "w", "p", "k", "q", "j", "x", "z", "or", "oh", "on", "os", "ow",
      "ok", "ox", "lb", "la", "li", "lo", "lm", "lx", "ft", "tv", "ti",
      "to", "dg", "de", "do", "dl", "du", "dm", "dj", "cd", "cl", "cm",
      "go", "gm", "ppm", "phi", "pdl", "be", "by", "kg", "kc", "kb",
      "km", "xi", "xu", "et", "eh", "en", "el", "em", "ew", "eq", "ex",
      "pya", "mph", "bpm", "wpm", "at", "ad", "ah", "ai", "an", "as",
      "am", "ax", "iv", "it", "id", "ii", "in", "is", "ix", "ne", "no",
      "nu", "sr", "sa", "so", "re", "hr", "hg", "he", "ha", "hi", "ho",
      "hl", "hm", "up", "kph", "us", "ux", "mr", "mg", "me", "ma", "mi",
      "ms", "ml", "mu", "mm", "my", "ye", "we", "vi"]
"philisophy" ["philosophy"]
"philosphy" ["philosophy"]
"phoptograph" ["photograph", "phonograph"]
"portay" ["portray", "portal", "sporty", "pray", "potty", "porgy",
         "party", "parlay", "poetry", "porter", "portage", "postal",
         "moray", "mortar", "mortal", "foray", "forty", "worthy"]
"practioner" ["practitioner"]
"precict" ["precinct", "predict", "relict", "precis", "prefect",
          "precept", "precise"]
"procdure" ["procedure", "procure", "produce", "brochure"]
"profesor" ["professor", "profess", "processor"]
"protoype" ["prototype", "protype"]
"punishmnet" ["punishment"]
"puppeter" ["puppeteer", "puppetry"]
"recuit" ["recruit", "remit", "recur", "deceit", "reduct", "rebut",
         "receipt",
         "recent", "recant", "recoil", "recuse", "result"]
"responsiblity" ["responsibility"]
"resposible" ["responsible"]
"revnue"["revenue", "revue", "revenge", "retinue", "venue", "rente",
```

```
        "avenue", "revere", "revise", "revile", "revive", "revoke",
        "rescue"]
"secetary" ["secretary", "sedentary", "nectary"]
"secreterial" ["secretarial", "secretariat"]
"seting" ["seating", "setting", "sting", "siting", "seeing", "sewing",
          "salting", "sheeting", "seeking", "seeming", "sitting",
          "sifting", "sorting", "sealing", "seizing", "testing",
          "stewing", "serving", "settling", "sending", "sensing",
          "selling", "eating", "string", "sing", "stung", "stint",
          "stink", "sling", "suing", "swing", "seine", "wetting",
          "meeting", "acting", "dieting", "doting", "saying", "saving",
          "satang", "skating", "serine", "tenting", "teeing", "outing",
          "netting", "renting", "rating", "retina", "heating", "siding",
          "sizing", "sweating", "letting", "luting", "ceding", "melting",
          "mating", "spring", "spying", "petting", "peeing", "beating",
          "betting", "belting", "being", "venting", "voting", "skiing"]
"sharholder" ["shareholder"]
"spokeman" ["spokesman", "spokesmen", "spoken", "spaceman"]
"stong" ["strong", "tong", "song", "sting", "stony", "stung", "stone",
         "sarong", "satang", "siting", "storing", "stowing", "string",
         "strung", "tog", "gong", "bong", "tang", "tony", "thong",
         "dong", "pong", "tung", "tonk", "tone", "hong", "long",
         "tons", "sing", "sang", "sung", "sone", "stag", "stop",
         "stow", "wrong", "atony", "atone", "along", "among", "stint",
         "stink", "stand", "stank", "stomp", "stout", "scone", "stunt",
         "stunk", "stoat", "stoic", "stood", "stoop", "stool", "story",
         "stork", "store", "storm", "stole", "stock", "prong", "shone",
         "slog", "sling", "slang", "slung", "suing", "smog", "swing",
         "swung", "stove", "stoke"]
"subcommitee" ["subcommittee"]
"surival" ["survival", "rival", "urial", "urinal", "arrival", "shrivel",
           "surgical", "serval", "serial", "surreal", "burial"]
"temporaray" ["temporary"]
"territority" ["territory", "territorial"]
"ting" ["sting", "ring", "bing", "tong", "tung", "tang", "tint", "tiny",
        "thing", "ding", "ping", "king", "tying", "tine", "sing", "ling",
        "wing", "zing", "eating", "rating", "aging", "mating", "siting",
        "icing", "outing", "doting", "voting", "owing", "luting",
        "using", "taint", "taping", "taking", "teeing", "tiling",
        "timing", "toying", "tuning", "tubing", "tinge", "tinea",
        "string", "stung", "stint", "stink", "sling", "suing", "swing",
        "lying", "trig", "trine", "tsine", "eng", "ink", "inn", "tug",
        "teg", "tag", "tog", "tit", "tid", "tic", "tip", "tie", "tin",
        "til", "going", "gig", "gong", "gang", "rig", "rung", "rang",
        "rind", "rink", "bring", "wring", "fling", "fig", "fang", "find",
        "fink", "fine", "fins", "being", "big", "bong", "bung", "bang",
        "bind", "bine", "yang", "hying", "dying", "vying", "vine",
        "vino", "tony", "thong", "tonk", "tone", "tons", "tune", "tuna",
        "tent", "tend", "tanh", "tank", "twang", "time", "doing",
        "think", "thine", "typing", "dig", "dong", "dung", "dang",
        "dint", "dine", "cling", "pig", "pong", "pung", "pang",
        "pint", "pink", "pine", "kiang", "kind", "kink", "kine",
        "kina", "kino", "jig", "jinx", "trag", "thug", "tier",
        "titi", "twig", "twine", "oint", "nine", "tire", "tiro",
        "tidy", "tide", "tilt", "tile", "till", "tick", "tift",
        "tiff", "sign", "song", "sung", "sang", "sinh", "sink",
        "sine", "hong", "hung", "hang", "hint", "hind", "long",
```

```
        "lung", "lint", "link", "line", "mig", "mung", "mint",
        "mind", "mink", "mine", "mina", "mini", "wig", "wang",
        "winy", "wind", "wink", "wine", "wino", "zig", "zinc", "tipi"]
"tomorow" ["tomorrow"]
"tranlator" ["translator"]
"unjury" ["injury", "jury", "usury", "unary", "injure", "unruly"]
"voluntarty" ["voluntary"]
"whatesoever" ["whatsoever", "wheresoever"]
"whather" ["whether", "whither", "weather", "whatever", "heather",
           "hater", "gather", "bather", "hither", "father", "hatter",
           "rather", "lather", "water", "wether", "wither", "washer",
           "whaler", "shatter", "leather", "thither", "chatter",
           "feather", "blather"]
"wherby" ["whereby", "hereby", "derby", "whey", "where", "sherry",
          "wheezy", "thereby", "cherry"]
"wome" ["woe", "some", "woke", "tome", "wore", "home", "dome", "come",
        "wove", "womb", "awoke", "woman", "women", "swore", "wrote",
        "worse", "wormy", "wolve", "me", "one", "ore", "ode", "owe",
        "we", "wee", "wow", "woo", "won", "wop", "wok", "same", "sone",
        "soma", "sore", "sole", "acme", "roe", "rime", "roue", "rote",
        "rose", "brome", "rode", "role", "romp", "rope", "robe", "rove",
        "wine", "wise", "wire", "wide", "wile", "wife", "wimp", "wipe",
        "wive", "were", "wane", "wake", "ware", "wade", "wale", "wage",
        "wave", "wont", "wonk", "worm", "gnome", "whose", "whore",
        "love", "yoke", "yore", "toe", "time", "tame", "tone", "tote",
        "tore", "tomb", "ooze", "name", "none", "note", "nose", "node",
        "whee", "wood", "wool", "woof", "word", "wort", "worn", "work",
        "hoe", "heme", "hone", "hose", "homo", "hole", "hope", "hove",
        "doe", "dime", "dame", "done", "dote", "dose", "dole", "dope",
        "dove", "doze", "came", "cone", "coke", "coma", "cote", "core",
        "cyme", "code", "cole", "cope", "cove", "comb", "mime", "moue",
        "mote", "more", "mode", "mole", "mope", "move", "foe", "fame",
        "fume", "fore", "wold", "wolf", "poem", "pone", "poke", "pose",
        "pore", "pole", "pomp", "pope", "game", "gone", "gore", "bone",
        "bore", "bode", "bole", "bomb", "vote", "vole", "joke", "zone"]
```