Department of Computer Science,
University of Otago

Technical Report OUCS-2005-06

Diagnosing and responding to student errors in a dialogue-based computer-aided language-learning system

Author:
Edwin van der Ham
Visiting student, University of Twente

Status: Project write-up

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.html

Diagnosing and responding to student errors in a dialogue-based computer-aided language-learning system

Edwin van der Ham

May 6, 2005

Abstract

The task of a Computer Aided Language Learning (CALL) system is to assist students in learning a new language. This is a complex task which requires amongst other things error detection and error correction techniques to detect and respond to the students' mistakes. Te Kaitito is a dialogue-based CALL system, primarily designed to teach the Māori language to native speakers of English. This report describes a new module developed for this system that uses perturbation of the input sentences entered by the user to diagnose errors in these sentences. Perturbations which result in sentences which the system can parse and/or which are more plausible in the given context than the original sentence are identified as possible corrections of the user's sentence. The technique of perturbing input sentences is combined with other techniques to make it more efficient. The information gathered by the module is then used to give feedback which will give the student a better understanding of his error.

Contents

Chapter 1

Introduction

1.1 Background: The Te Kaitito system

The task of a Computer-Aided Language Learning (CALL) system is to assist a student in learning a new language. Te Kaitito is a dialogue-based CALL system, developed at Otago University. The system has a modular architecture and can be configured to teach any language for which a grammar and lexicon have been developed. Currently, research is focused on the tutoring of the Māori language to English native speakers, but in theory any language can be taught to native speakers of any other language. In my research, I used an English grammar and lexicon; the system is therefore assumed to be directed at learners of English.

A key aspect of the system is that it is dialogue-based. This allows the system to interact with the student in a human-like fashion. The system is able to remember facts it has been told by the student and can ask the student questions about these facts later on. It can also ask 'genuine' questions about the student to find out more about him or her and it can tell the student facts about itself. The best way to explain how the system works is by giving an example. Figure 1.1 gives a typical example of an interactive dialogue between a student and the system. In this example the English grammar is used, so we assume a scenario where the student is a learner of English. As long as the student tells the system (new) facts or asks the system questions, the system will either accept the fact or answer the question. The student can also leave the initiative to the system, simply by hitting 'return', which will trigger the system to ask the student a question or make an assertion.

TK	How are you? (1)	TK	I am John.
S	I am fine.	S	
TK	Okay.	TK	Where are you from? (6)
S	How are you? (2)	S	I am from Dunedin. (7)
TK	I am fine.	S	
S	(3)	TK	I am also from Dunedin. (8)
TK	What is your name? (4)	S	
S	My name is Ann.	TK	Let's see if you remember: Who am I? (9)
TK	Okay.	S	You are John.
S	Who are you? (5)	TK	Correct, well done! (10)

Figure 1.1: A simple mixed-initiative dialogue (from Slabbers and Knott [11])

In the example, the system takes the initiative (Turn 1) by asking *How are you?*. The student tells Te Kaitito it is fine and the system lets the student take the initiative. He decides to ask the system a similar question (Turn 2). Then the student concedes the initiative (Turn 3) and the system asks another question (Turn 4). The student answers this question and decides to ask the system who it is (Turn 5). When the student lets the system take the initiative again, the system has another question for the student (Turn 6).

This question is in turn answered (Turn 7) and the system generates a new assertion on the current topic (Turn 8). Finally, when the student decides to concede the initiative again, the system generates a teaching question (Turn 9). The student answers this question correctly and receives positive feedback (Turn 10).

1.2 The goal of the project: Input error correction

The preceding example of a dialogue with Te Kaitito is a good example for the case that the student has done his homework well. He knows what grammatical constructions to use and how to spell the words correctly. Even more he seems to understand the questions being asked by the system and is able to answer them correctly. Unfortunately this is an ideal situation. Because the student is learning a new language, he is prone to make a lot of errors in his utterances. Some of these errors might be typos or misspellings of words. More severe errors might involve grammatical errors or even completely incorrect words. Te Kaitito is able to deal with all these errors in a very basic way. Its solution is to create a response which tells the student that it was unable to understand what he meant. However, this does not help the student much in understanding what mistake he made.

The goal of my project is to give the system some tolerance to errors in the user's utterances. Specifically the goal is to create a module which creates variations on a sentence which cannot be interpreted and tries those out instead. These variations are called perturbations, after a suggestion by Lurcock [9].

1.3 Overview of the perturbation module

My contribution to this system comprises a module that uses perturbation to diagnose errors in the input given to the system. Before the development of this module, the system had no way of handling input errors other than recognizing that the input is incorrect and informing the student about this. Te Kaitito is a tutoring system and is supposed to be a good alternative to learning a new language from a real teacher. A teacher would not just inform the student that his input is incorrect, he would analyze the input and try to find a cause for the mistake. This is what Te Kaitito should do as well. I designed and implemented a perturbation module which accomplishes this task.

In general, the module consists of three separate components. The **general perturbation component** takes the input sentence and creates a set of perturbations of this sentence. Perturbations are made on both a single character and a whole word level. The creation of character level perturbations corresponds to the way a spellchecker works. A spellchecker bases the best word replacement candidates on the closeness of a word to the incorrect word. The smaller the number of differing characters, the more likely the suggested word would be. Word perturbations are more complicated. To get good results from this, we need a model of student errors. Every character and word level perturbation is given a score to preselect more likely perturbations. Only the most likely perturbations will be passed on to the sentence interpretation system.

The **perturbation selection component** is the process whereby the original sentence plus the set of likely perturbations are all processed by Te Kaitito's sentence interpretation system. This is already designed to handle ambiguous sentences and to select the best available interpretation using many sources of information. Adding perturbations simply constitutes more source of ambiguity. At the end of this process, one or more interpretations survive.

If the surviving interpretations come from a perturbation of the original sentence, the system has effectively diagnosed an error. The **feedback component** is responsible for giving the correct teaching feedback. As I pointed out before, the system is supposed to help students in learning a language, similar to the way a teacher would. The way in which the system provides the student with feedback is therefore very important. However, this was not the focus of my project, so I have implemented a very simple scheme that can easily be extended. The general perturbation component provides a lot of information about the student's mistake, so a more sophisticated feedback system can rely on this information.

1.4 Structure of this report

In chapter 2 of this report, I will review some literature on the subject of CALL systems and error detection and correction techniques. Also, I will discuss Te Kaitito, the CALL system in subject. Chapter 3 gives a classification of perturbation types. This classification gives ground to the design of the perturbation algorithm which I present and discuss in detail in chapter 4. In this chapter, the need for a backoff algorithm is discussed. I will go into detail on this backoff algorithm in chapter 5. Chapter 6 presents the results of integrating the perturbation module into the system. Some interesting dialogues are given, augmented with an explanation of the system's choices in interpreting the student's input. Finally, in chapter 7 I will summarize my findings and give some suggestions on what needs to be done to improve the system's performance.

Chapter 2

Literature review

In this chapter, I review the background literature relevant to the project. In section 2.1 I describe Te Kaitito. Section 2.2 gives a summary of existing spellchecking techniques. In section 2.3 I focus on systems that have been developed specifically for correcting student errors in CALL systems. Finally in section 2.4, I describe how this all comes together into a nice framework for implementing a perturbation module.

2.1 The Te Kaitito system

The system for which I developed and implemented a perturbation module is Te Kaitito, a multi-language dialogue system that aims to assist students in learning a new language. The system has a modular design which means that new functionality can be integrated into the system without having to change its main architecture. The system has a few different interfaces including one that operates via a webpage and one that accepts input from a speech synthesizer. The one I use is a simple text-only interpreter inside a lisp session. A student can give input to the system by entering a sentence and the system responds by printing text on the screen. An example of such an interaction is already given in figure 1.1.

After the student entered a sentence, the sentence is parsed using the LKB system [2]. The parse result is sequentially acted upon by a number of modules. Every module can extract information from the parse and pass it on to the next module. The modules are arranged as a pipeline. At every stage in the pipeline, there is potentially ambiguity in the information that is extracted. In this case, all possible interpretations are passed forward to the next module. There is thus the potential for a very large number of interpretations at the end of the pipeline. When the parse reaches the end of the pipeline, every module along the line has gathered useful information about the sentence. This information is combined to determine one single interpretation of the original sentence. If there are still multiple interpretations left, the system will ask the student for clarification. If no interpretation can be constructed, the system can do nothing but tell the student that it could not understand him.

2.1.1 Te Kaitito's utterance interpretation pipeline

Along the pipeline, conceptually there are five functional modules. First, the initiative module takes its turn. This module will take the initiative when the student inputs an empty line (see Slabbers and Knott [11] for details). Then, if the student has entered a sentence, the sentence is given to the parsing module, which derives a set of parse trees, each with an associated semantic representation. This semantic representation contains referring expressions, some of which involve presuppositions which must be resolved using the current dialogue context. For instance, the referring expression *the dog* is presuppositional; it refers back to an existing object in the dialogue context, and we need to find out which one it is. Finally there is a module to determine the type of the dialogue act which can either be a question, the answer to a question, an assertion or a clarification question. At the end, based on the information gathered by the preceding modules, a disambiguation process takes place to try and come up with only one single interpretation of

the sentence. Take for example the following dialogue:

A: The big fruit flies ate the fresh fruit
B: OK
A: I threw the mouldy fruit at the small fruit flies
B: How does the fruit fly?
A: The fruit flies like a banana

The last sentence is ambiguous on every level. It is syntactically ambiguous because it can be a sentence about fruit that can fly or it can be about flies which happen to be fruit flies. It is ambiguous in terms of referents, because in the recent context there has been fresh and mouldy fruit as well as big and small fruit flies. As a dialogue act, the sentence is ambiguous because it can either be a new assertion or the answer to the question that has just been posed. Figure 2.1 shows the interpretations of this sentence that are constructed by the different modules in the pipeline. Note that because of all this ambiguity there are eight different interpretations of this one single sentence.

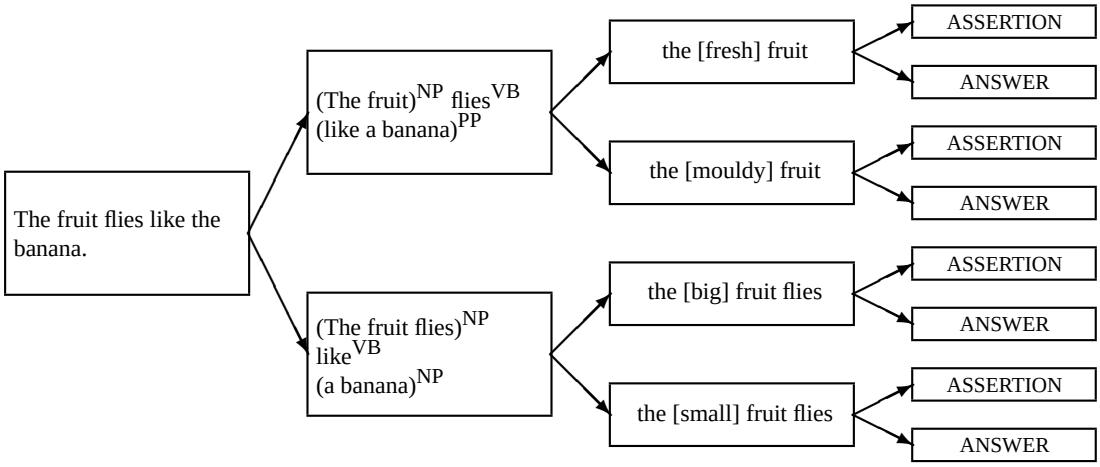

Figure 2.1: A complete set of possible interpretations on every stage in the pipeline. (from Lurcock [9])

2.1.2 Te Kaitito's disambiguation algorithm

The disambiguation module is quite sophisticated. It needs to make a profound choice between the eight interpretations. To do this, information gathered by the preceding modules is needed. Lurcock [9] argues that the disambiguation process should consult the information gathered by the modules in the opposite direction. He suggests that dialogue-act level information is stronger than information on all other levels. For instance, if the system asked the user a question and only one interpretation of the user's response counts as an answer to that question, this interpretation should be preferred, regardless of preferences at lower levels. The same thing holds for the semantic and syntactic levels. So the disambiguation process starts by looking at the dialogue act. A question has been posed about fruit, so the sentence should be interpreted as an answer to a question about fruit. Pruning the lower half of the complete set of interpretations, this leaves only two possible interpretations. (figure 2.2)

The two remaining interpretations will (hopefully) be disambiguated using semantic information about the aforementioned fruit. In the example, since the mouldy fruit was mentioned more recently than the fresh fruit, the system will choose mouldy fruit as the correct interpretation. This leaves us with just the single interpretation shown in figure 2.3.

If there remains more than one interpretation after the pruning process, Te Kaitito will ask a clarification question in order to find out what the intention of the speaker was. Clarification questions on the syntactic level need to be asked before clarification on a semantic level can be made. Opposite to the direction the disambiguation process works, the clarification process consults the different modules in the same order as

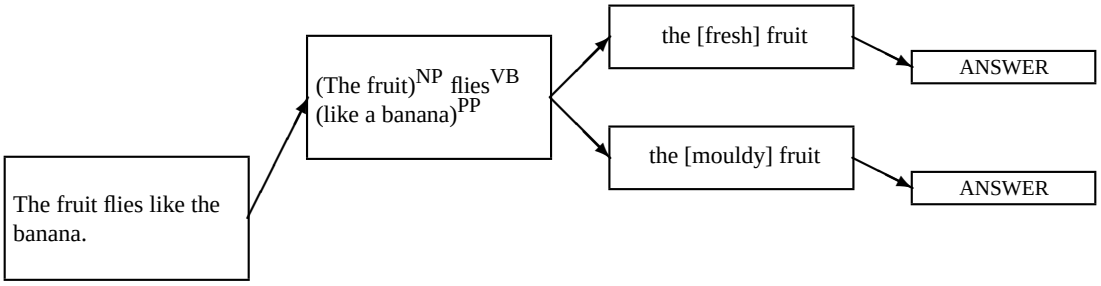

Figure 2.2: Possible interpretations remaining after dialogue-act filtering. (from Lurcock [9])

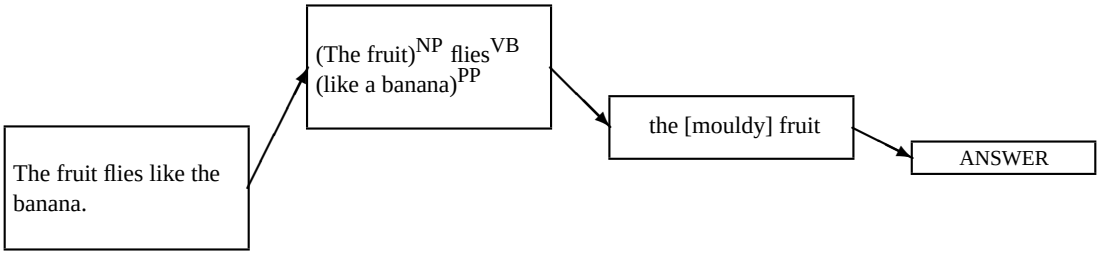

Figure 2.3: Final interpretation remaining after semantic disambiguation. (from Lurcock [9])

the initial interpretation process. Figure 2.4 schematically shows the process that takes place to interpret, disambiguate and eventually clarify an input to the system, so that one single interpretation is left.

2.2 Spellchecking systems

The remainder of this chapter discusses existing techniques for detecting and correcting errors in utterances. We begin by looking at spellcheckers.

Ever since the introduction of word processors, techniques for error correction have been developed. Even today, none of these techniques are perfect and a lot of work still needs to be done. In 1992, Kukich [8] wrote a paper called 'Techniques for Automatically Correcting Words in Text'. This paper gives a very good summary of the current state of affairs in error correction techniques. To begin with, we assume that some process has introduced errors into a sentence. This process could be Optical Character Recognition (OCR), speech interpretation or a fallible human writer. For CALL systems like Te Kaitito, spelling correction might be very useful in the detection of errors and correction of the input to the system. I will give a short summary of a few chapters in Kukich's paper.

2.2.1 Detection versus correction

An important distinction to be made is between error detection and error correction. Detection of errors can be done by matching words with a dictionary or alexicon or it can be done using n-gram techniques on sequences of characters. Error correction is much harder, new (buzz) words are introduced every day and in English, almost any noun can be verbified. The task of error correction can be split up into three increasingly difficult tasks, i.e nonword error detection, isolated-word error correction and context-dependent word correction. I will discuss these three topics in turn.

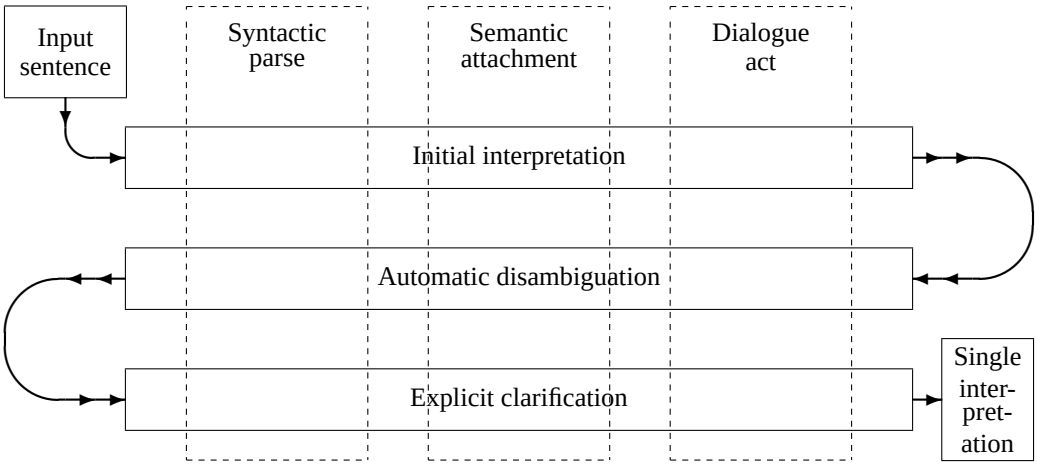

Figure 2.4: The interpretation process viewed as a pipeline.

2.2.2 Nonword error detection

Nonword error detection research has focused on n-gram analysis and dictionary lookup. N-grams are subsequences of a word or string containing exactly n letters. The parameter n is usually very small (two or three). N-gram error detection works by looking up an n-grams from the input sentence in a precompiled table of n-gram statistics. This table contains information about the frequency or existence of any n-gram in a typical text. If an uncommon n-gram is found, it is assumed to be an error. It requires a large corpus of text to be analysed to compile a lookup table.

The other technique is dictionary lookup. This technique is very simple, every word in a text is looked up in a dictionary. If the word is not found in the dictionary, it is assumed to be incorrect. N-gram techniques have proven to be quite effective, but nonetheless dictionary lookup is still more effective in the detection of *human generated* errors.

A problem with both of these techniques is that many mistakes result in (new) real words, e.g. *The dog like the cat* is wrong if it needs to be interpreted, but all of the words in the sentence are genuine words, and will be found in the dictionary. Context-dependent word correction techniques are needed to resolve these kinds of errors. Section 2.2.4 deals with those.

2.2.3 Isolated-word error correction

As for the *correction* of errors, lookup of words or n-grams is not enough. To be able to correct words, information is needed about spelling error patterns which are usually very task specific. A distinction in patterns can be made between typographic, cognitive and phonetic errors. Typographic errors occur when the writer knows the correct spelling of a word, but simply makes a mistake in typing it. These keyboard effects play an important role in computer-aided applications. Cognitive errors are formed by a lack of knowledge on the part of the writer. Phonetic errors are actually a subclass of cognitive errors in which the writer substitutes a sequence of letters by a phonetically similar one.

On another level, different error types can be distinguished. Damerau [3] found that 80% of all misspelled words contain a single instance of one of the following four error types: insertion, deletion, substitution and transposition. This notion is very important, for it allows for error correction systems to focus on a single error per word. However, the effect of the word length can make things harder. Even though spelling errors occur less frequently in short words, the number of possible corrections is much higher and correction of short words thus lead to miscorrections.

Techniques for resolving isolated word errors need to accomplish three tasks, i.e. detection of an error, generation of candidate corrections and ranking of these candidates. Detection of an error can be

accomplished using either n-gram techniques or dictionary lookup. The generation process usually involves replacing incorrect words or n-grams with a correct word or n-gram. Some metric for similarity can be used to rank the candidate replacements. Techniques used for both generating and ranking replacements are amongst other things minimum edit distance, rule-based techniques, n-gram based techniques, probabilistic techniques and (more recently) neural networks.

2.2.4 Context-dependent word correction

By far the most difficult to detect are errors involving only real words. Not much is known about the frequency of these errors in texts. The frequency definitely very much depends on the text source. A small study has been done by Atwell and Elliott [4] on samples of 50 errors from three different sources: published, manually proofread text, essays written by 11- and 12-year old students and essays written by non-native English speakers. The results show that the context-dependent word error rate is significantly higher in the texts written by non-native English speakers than in the other two categories. This points out that the detection of such errors is an important aspect in CALL systems.

A number of techniques for resolving these errors have been developed. One of the first ever used techniques is an acceptance-based technique. This technique assumes that errors can simply be ignored as long as some interpretation can be constructed that is meaningful to the application. The same reasoning serves as a basis for the error grammar technique discussed in section 2.3.1. A second approach is constraint relaxation. This technique is widely used in CALL systems, section 2.3 goes into more detail on systems developed for correcting student errors.

Another range of techniques that can be used are expectation-based techniques. In a system using such a technique, the system builds a list of words it expects to see in the next position. This approach involves statistical language modeling to compile a corpus of naturally occurring word tuples of n-grams from which expectations about the next word can be derived. Instead of looking up sequences of characters in a corpus, n-gram word techniques lookup sequences of words in a corpus. For character n-gram statistics, we already need a large corpus to get reliable results. For n-gram word lookup, the corpus needs to be even larger. If the corpus is not big enough, the chance of finding an exact match of a word n-gram in the corpus will be low. A backoff mechanism is needed to make this technique workable. A backoff mechanism for n-gram word techniques will be discussed in detail in chapter 5. In the next section I will give a short introduction to the concept of backoff.

2.2.5 Backoff mechanisms

A backoff mechanism is needed to improve the performance of word n-gram lookup techniques. One way of backing off is using lower order n-grams. We might start by looking up trigrams. If the word trigram is found in the corpus we can assign a probability to the trigram using the counts of the trigram in the corpus. But even if the word trigram is not found, we might still want to assign a probability higher than zero to the trigram. We can instead of looking up the trigram, back off to looking up a bigram which is part of the trigram. Katz [7] has designed a backoff algorithm which does this.

Another way of backing off is by tagging words with their most likely morphological class. The first level of backoff then involves tagging one word in the trigram. The trigram again is looked up in the corpus with the difference that for the tagged word, any word having the same tag will suffice. This will hopefully increase the number of trigram occurrences while still weeding out completely unlikely trigrams. The corpus of course needs to be tagged with correct morphological classes for this to work. The next backoff step involves tagging two words in the trigram instead of only one. The chances of finding some matching trigrams in the corpus will increase. When even two tagged words do not give evidence of the trigram being correct, the last backoff step would be to tag all three words in the trigram.

These two backoff mechanisms are both useful. We can even combine them so that we have got a backoff mechanism for the trigram word lookup method. Table 2.1 shows one way in which the two mechanisms can be combined. The top-left corner schematically shows a trigram corpus lookup. The string $W_1W_2W_3$ stands for a trigram lookup. On the horizontal axis the first backoff mechanism is applied to the original trigram, on the vertical axis the second. The second mechanism involves tagging words with their morphological class. This is denoted by the T instead of a W.

	trigrams	*bigrams*	*unigrams*
no tag	$W_1W_2W_3$	W_1W_2 W_2W_3	W_2
one tag	$T_1W_2W_3$ $W_1T_2W_3$ $W_1W_2T_3$	T_1W_2 W_2T_3	 T_2
two tags	$T_1T_2W_3$ $T_1W_2T_3$ $W_1T_2T_3$	T_1T_2 T_2T_3	
three tags	$T_1T_2T_3$		

Table 2.1: A combined trigram lookup backoff-mechanism

2.3 Systems for correcting student errors

Research has already focused on error correction in CALL systems. The aim of a CALL system is to help the student in learning a new language. Any utterance that the language learner makes somehow needs to be interpreted to enable the CALL system to provide the student with useful feedback. The student is prone to make grammatical errors and robust parsing techniques are needed to assign an interpretation to (grammatically) incorrect sentences. At the same time, the system needs to determine the origin of the mistake in order to give a response that will help the student in understanding his mistake.

Two robust parsing techniques that are widely used are a technique that makes use of error grammars and a constraint relaxation approach. I will dicuss both techniques in some more detail.

2.3.1 Error grammars

Student errors in a language learning environment are different from general errors found in texts. The errors that students make tend to come from incorrectly applying grammatical rules thus producing grammatically incorrect sentences. A technique for resolving such errors can do exactly the same. By incorrectly applying grammatical constructions, an extra set of rules is created. These rules are called mal-rules or error rules. If we incorporate the error rules in a rule-based parsing system, the system can handle much more constructions and is therefore able to assign an interpretation to incorrect sentences while still being able to tell whether the sentence is correct or not. This information can be used to correct and give feedback on student utterances.

Bender et al. [1] describe an error correction approach using a rule-based technique. They implemented a system capable of producing well-formed semantic representations for ill-formed input. This system makes use of the same typed feature grammar our system uses, which they augment with mal-rules. These rules assign a semantic representation to grammatically incorrect sentences. The assigned representation is the representation that would be given to the most likely correction for the particular rule.

2.3.2 Constraint relaxation

Another approach to the detection of student errors is to (temporarily) ignore constraints in the grammar instead of introducing extra rules. Grammar rules usually have a number of constraints associated with them. When ignoring some of the constraints, a weaker instance of a grammar rule is created that, when applied to the erroneous student input, may produce a result where the original rule does not.

Menzel and Schroeder [10] propose an integrated approach to robust parsing and error diagnosis using graded contraints on multiple levels of knowledge. The underlying technique they use is a technique called eliminative parsing. Instead of constructing a syntactic interpretation by building complex structures out of elementary building blocks, eliminative parsing works the other way around. The process starts considering all possible dependencies between the words in a sentence. From this point on, a number of constraints on these dependencies are applied, until one single dependency structure is left corresponding with an interpretation of the sentence. This technique is not robust for it fails to construct an interpretation for

ungrammatical sentences. However, it is possible to find the interpretation which has the least constraint violations. This information can be used to determine the kind of mistake the student made, providing him with correct feedback.

2.3.3 Error grammars versus constraint relaxation

Bender et al. [1] compare their error grammar approach to the constraint relaxation technique by Menzel and Schroeder [10], who argue that adding mal-rules to a grammar is an ineffective way to approach the problem of grammar checking for CALL systems. The relaxation approach may be more robust than the mal-rules approach, nonetheless they have two arguments as to why their approach is still interesting. First, a grammar can be made more precise by adding particular mal-rules as opposed to removing constraints which make the grammar less precise. Second, a system which can accurately and helpfully diagnose some common errors already benefits language learners. The question is not which technique has the best error coverage but which technique has the highest value to the language learner. It can be argued that a more correct error correction system has preference over a less correct system with a higher error coverage.

2.4 Combining syntactic, semantic and disambiguation information

If we want tutoring systems to perform at the level of a human teacher, their understanding process needs to mimic human language understanding processes. A key aspect in language understanding processes is that we are capable of combining all kinds of information to construct a correct interpretation of a sentence. This means that not only grammatical knowledge, but also contextual information and even some common sense knowledge is used to perform the task.

An early example of a system which uses multiple levels of interpretation if that of Hobbs [5]. Hobbs' system processes a sentence at the syntactic, semantic and discourse levels, and uses information from all these levels to disambiguate. Hobbs proposes that an error correction mechanism should also be able to make reference to all these levels, so that when correcting the error, it can pick the correction which is most likely at all of these levels.

Menzel and Schroeder [10] recognise this too and try to adopt this kind of reasoning in their approach to robust parsing. I already mentioned that they use the constraint relaxation technique on multiple levels of knowledge. The levels on which the system works are the syntactic, semantic and domain knowledge level. They find that this is very robust and error diagnoses can easily be extracted from the parsing result for ill-formed input.

This multi-level model already exists in Te Kaitito, as already described in section 2.1. The pipeline incorporates a syntactic, semantic and dialogue-act level. The only thing it does not incorporate yet is a *robust* parsing system. We could adapt the existing parsing system to have it return partial parses from which to reconstruct the intended utterance. An easier way of incorporating robustness is by including a pertubation module which generates variations on the original sentence using various error correction techniques. Lurcock [9] points out that such a module would fit very nicely into the architecture of Te Kaitito. Figure 2.5 shows his proposed augmented interpretation pipeline.

The perturbation module should operate on the raw sentence even before the syntactical parsing takes place. Determining which is the best correction is then wholly subsumed within the normal utterance disambiguation process. If a corrected version of the input sentence scores higher on the dialogue act level, it should straightforwardly be preferred. Likewise, if it scores higher on the semantic level, it should be preferred. If two perturbations score equally well on the dialogue act and semantic levels, we may be able to distinguish between them by comparing how likely the two perturbations are, again drawing on techniques for error correction. If they are equally likely, then we have to ask a clarification question to the user to see which version they meant. Thus adding a perturbation module simply involves adding an initial module to Lurcock's disambiguation architecture as shown in figure 2.5.

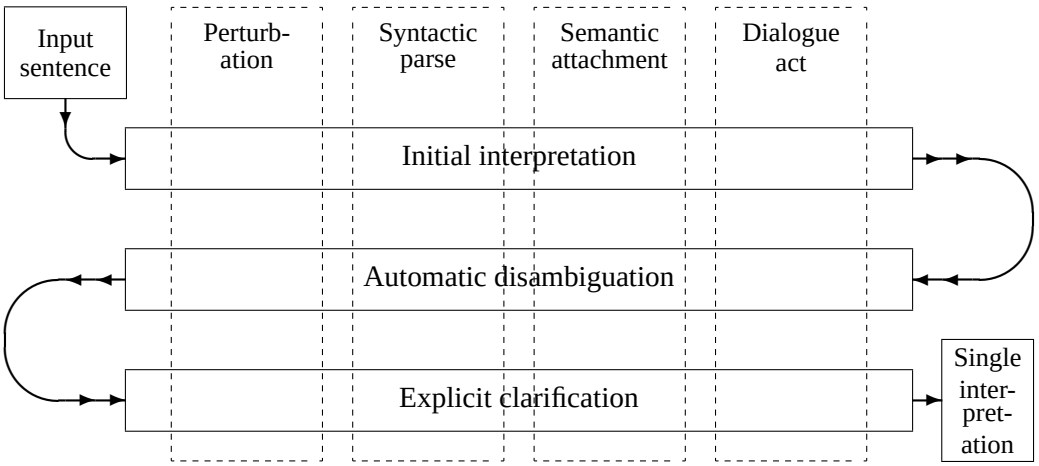

Figure 2.5: The interpretation pipeline including the perturbation module. (from Lurcock [9])

Chapter 3

A classification of perturbation types

To start perturbing sentences, we need a mechanism to create the perturbations. There are numerous ways of perturbing a sentence and there is no point in making random perturbations. A lot of research has already been done on the subject and results of this research can be used to classify perturbations. This classification helps in deciding how to make the perturbations.

3.1 Perturbation types

To be able to indentify and classify perturbation rules, one should first try and determine the kind of errors commonly made by people. In section 2.2.3 the important fact is given that 80% of all misspelled words contain a single instance of one of the following four error types: character insertion, character deletion, character substitution and character transposition. An insertion error in this context occurs when the student inserted an extra character where there should be no extra character. Similar reasoning holds for deletion, substitution and transposition errors. There is a subtle difference between substitution and transposition, namely in substitution a character is replaced by one that should not be there at all whereas in transposition characters are only mixed up. A typical input sentence can contain more than one misspelling, but as Kukich found, this is quite unlikely. It seems like a good idea to take these four error types as a basis for constructing perturbation rules.

3.2 Perturbation level

For the purpose of spellchecking, the four error types apply to errors that are found on a character level. By deleting, inserting, substituting or transponing one character in a word, a spelling error can be corrected. But how about grammatical errors, those can usually not be corrected by changing a single character in one word. Grammar is to do with how words are combined together into sentences. Therefore, grammatical errors usually involve syntactically correct words and parsing or other specialized techniques are needed to detect the errors.

On the other hand, grammatical errors might be classified using the same four perturbation types. The only difference is that we do not apply the types to single characters but to complete words. An insertion error now means that the student inserted a complete new (correctly spelled) word which makes the sentence grammatically incorrect. Similar things can be said about the other three perturbations types. This introduces a second level on which to make perturbations. The first level is the character level and the second level is the word level.

3.3 Summary of the varieties of perturbation

Now that we have distinguished two dimensions in which to look at perturbations, the next step is to make a table containing the two dimensions. For every type of error and level at which it occurs, an example

can be given. Furthermore, it is probably the case that every single table cell needs a particular method for resolving the error. Every cell in the table can be filled in with both a typical example of the kind of error to be expected and a hint to which method would give the best results for disambiguating the sentence containing the error. Table 3.1 shows this table. In sections 3.4 and 3.5 I will go into more detail on the different perturbation types.

		Level	
		Character level	*Word level*
Perturbation type	*Insertion error*	Insertion typo e.g. ”party” → ”pasrty” Doubling insertion e.g. ”very” → ”verry”	Word insertion error e.g. ”I like nature → ”I like the nature”
	Deletion error	Deletion typo, e.g. ”party” → ”paty” Doubling deletion e.g. ”blurry” → ”blury”	Word deletion error e.g. ”John likes the dog” → ”John likes dog”
	Substitution error	Keyboard substitution typo e.g. ”party” → ”psrty” Phonological substitution e.g. ”party” → ”parti”	Word substitution error e.g. ”I like his dog” → ”I like him dog”
	Transposition error	Transposition typo e.g. ”party” → ”patry”	Word transposition error e.g. ”I saw a big dog” → ”I saw a dog big”

Table 3.1: Classification of common errors

3.4 Character level errors

Errors at the character level are fairly easy to detect and correct and usually we are able to determine what kind of mistake caused the error. Where this is not possible, we can usually still correct the error anyway. In this section, I discuss the four different character level errors.

3.4.1 Character insertion error

An insertion error occurs when an extra character is inserted that should not be there at all. The most common reason for the insertion of an extra character is related to typing. It often occurs that somebody hits two keys on the keyboard at the same time, causing an extra character to be inserted when there is no intention of doing this. If this is the case, the character being inserted must on the keyboard be spatially close to the next or previous character in the word. This kind of error is called an **insertion typo**.

When the inserted character is not spatially close to the next or previous character in the word, it is less likely to be a typing error. Another possibility is that the student made a grammatical error in wrongly doubling a consonant. The student is likely to do this when the doubling occurs in his native language. In some cases a doubling of a vowel can occur as well. We call this kind of error a **doubling insertion**.

If the insertion is close on the keyboard nor doubling related, it is unclear what the cause of the error is. The most likely cause for such an error is that the student just used an incorrect word. This kind of error

is simply called a **character insertion error**. A single character insertion error is very unlikely to occur in language learning applications.

3.4.2 Character deletion error

A deletion error occurs when a character is missing in a word for it to be correct. This could also be related to a typing mistake. The student could have been typing very fast causing him to miss one letter in a word making a typo. However, this is not very likely. Much more likely is that the student omitted doubling of a consonant. This can only happen when the foreign language that the student is trying to learn needs doubling where the student did not expect this. This error is called a **doubling deletion**. There is no way of determining if a **character deletion error** is actually a typo or not.

3.4.3 Character substitution error

A character substitution error occurs when a character is omitted and replaced by another character. This again is likely to be a typing error. If the student is typing very fast and accidentally hits the wrong key, a substitution error took place. Like in the case of an insertion typo, the substitution character should be close on the keyboard to the character being substituted. As expected we call this a **substitution typo**. If this is not the case, then it probably was no typo.

Another possibility for a substitution to take place is when the student makes a spelling mistake. Table 3.1 gives an example of such a **phonological substitution**. Since the error has a phonological origin, I should note that such a mistake is only made when the interchanged letters are close in sound. Close in sound are (in English) consonants like *t* and *d* and vowels like *a* and *e*. Some languages, like Māori, make extensive use of macrons. If you consider letters with macrons to be different letters, then these become a special case of closeness in sound. Furthermore, the probability of a phonological substitution error increases for it is very likely for a student to interchange the characters *a* and $\bar{a}$.

As before, if the substitution is not related to typing and has no phonological cause, it is unclear why the error occurred and we simply speak of a **character substitution error**.

3.4.4 Character transposition error

When it comes to transposing characters, it is not very likely to be anything else than a typo. There is no reason for a student to transpose two characters in word because the characters are somehow close in sound or something else. If a **character transposition error** is detected, it is reasonable to assume the student made a typing error.

3.5 Word level errors

While errors at the character level might be caused by spelling mistakes, word level errors certainly are not. Errors at the word level are clearly an indication that the student did not fully understand the utterance or its grammatical structure.

3.5.1 Word insertion error

A **word insertion error** indicates that the student inserted an additional word into the sentence. The most likely cause for this error is found in the native language of the student. The sentence might be a word-to-word translation from the speaker's native language which almost correctly constructs the correct utterance. The detection of a word insertion is fairly simple because the student introduced a word that should not be there.

3.5.2 Word deletion error

A word deletion is very similar to a word insertion. Again, the most likely cause is the word-to-word translation of a sentence in the student's native language. Only this time, the student omitted a word instead of inserting one. Detection of a **word deletion error** is more difficult because it involves introducing seemingly arbitrary new words to reconstruct the correct sentence. Information about the grammatical structure of the sentence is needed as well.

3.5.3 Word substitution error

A **word substitution error** occurs when a word is replaced by a new word in the same or a related morphological class. This kind of error gives a strong hint that the student made a grammatical error. Detection of a substitution error could be tricky when the substitution constructs a grammatically correct sentence with a different meaning. Especially in Māori, chances of such a substitution are quite high.

3.5.4 Word transposition error

A **word transposition error** indicates that the student made an error in the grammatical structure of the sentence. Word order rules are different between languages and are likely to cause transposition errors.

Chapter 4

The perturbation algorithm

In his chapter about dealing with errors in the Te Kaitito system, Lurcock[9] already discusses how a perturbation module can be integrated in this system. He explains that the disambiguation framework conceptually consists of a single pipeline that the sentence is fed into. The pipeline is shown in Figure 2.5. It contains in this order a syntactic parser, a semantic parser and finally a dialogue act module. The perturbation algorithm fits at the very beginning of the pipeline because perturbation operates at a very low level of characters and words and is not concerned with syntactic features. Every perturbation should concurrently be fed into the pipeline producing a significant number of extra interpretations of the original sentence.

When the disambiguation process is needed to select a particular interpretation, the last step should be to disambiguate by using the perturbation results. If however the other modules fail to determine the correct interpretation with reasonable certainty, the perturbation module should be able to evaluate the different perturbations. Lurcock points out that the last resort clarification step would be easy to to implement for the perturbation module. The different perturbations can simply be presented to the user as they are, trying to differentiate between them.

4.1 Global structure

From this point on, I will use some pseudocode to explain the structure of the perturbation algorithm. The code is meant to be self-explanatory, but for the sake of completeness I added some comments to the code which are always preceded by '%%'. Words surrounded by round brackets denote a structure, words surrounded by square brackets are (sequential) lists of elements denoted by that word. The function 'add' is assumed to add a new element to end of a list, the name of a list preceded by a number between square brackets (e.g. input[3]) denotes a particular element in the list. The notation 'list[x..]' denotes the list containing all elements of the original list from element x. The first element of a list is assumed to be element 0. Finally, the '+' operation on a list denotes either the addition of a new element to the end of the list or the concatenation of two lists.

The global structure of the perturbation algorithm looks like this:

```
main (input-string) {
  %% data structures for storing word and character sequences
  [word] sentence;
  (sentence, type, penalty) perturbation;
  [perturbation] perts;

  sentence = tokenize(input-string);
  perts.add(make_char_perturbations(sentence));
  perts.add(make_word_perturbations(sentence));
  perts = rank_and_select(perts);

  return perts;
```

```
}
```

The input for this function is a sentence represented as a string, i.e. a sequence of characters. The data structures defined are a list of words which will contain the words in the original sentence, a structure called 'perturbation', used for storing a particular perturbation of the sentence as well as the perturbation type and an associated penalty. The structure called 'perts' contains a list of those perturbations. First of all, the input is tokenized, i.e split into a list of words. This is needed because we want to create perturbations of single words. This list of words is given to a function that returns a list of different character level perturbations as well as a function that returns a list of different word perturbations. The perturbations are all added to the global perturbation structure, which is returned at the end so it can be used by another module. Before returning the structure, the 'rank_and_select' function is called to select only the most likely perturbations even before they are fed into the pipeline.

The 'tokenize' function is very straightforward. It takes a character sequence and splits this into multiple sequences, every sequence containing exactly one word. A new word is assumed to start at the start of the original sequence as well as every time after a space character is encountered. Also, the last word in the sentence, which is obviously not preceded by a space, is added before the list of words is returned.

```
tokenize(input) {
  [word] sentence;
  int start = 0;

  %% collect all space-separated character-sequences (words)
  for (i=0 to length(input); i++) {
   if input[i] = ' ' then
    sentence.add(input[start..i]);
    start = i+1;
  }
  sentence.add(input[start..]);
  return sentence;
}
```

4.2 Perturbation functions

After having split the input into separate words, the two main perturbation functions are called. The perturbation functions create a number of candidate perturbations. The way in which the candidates are created depends on the level of perturbation. Therefore the character and word level perturbations are treated separately.

4.2.1 Character level perturbations

As I explained in chapter 3, there are four different kinds of perturbations. For every word in the sentence, every kind of perturbation will be made. As there are a limited number of characters in a language, it is feasible just to try out every perturbation of every word in the sentence. A lot of perturbations can be weeded out by spellchecking the newly created word. We can get rid of even more perturbations by applying a penalty to every perturbation based on the likelihood of the particular perturbation. The number of perturbations that will survive this selection mechanism needs to be small, because parsing a sentence is a very expensive operation in terms of computing power and therefore time. The pseudo code for this function looks like this:

```
make_char_perturbations(sentence) {
   [sentence] parselist;
   (sentence, type, penalty) perturbation;

   for (i=0 to length(sentence); i++) {
```

```
      perturbation.add(char-insertion(sentence[i]),
                              "char-insertion", char_insertion_penalty);
      perturbation.add(char-deletion(sentence[i]),
                                "char-deletion", char_deletion_penalty);
      perturbation.add(char-substitution(sentence[i]),
                             "substitution", char_substitution_penalty);
      perturbation.add(char-transposition(sentence[i]),
                           "transposition", char_transposition_penalty);
   }
   return perturbation;
}
```

For making the character perturbations, the same perturbation structure as the one in the main function exists. The 'add' function that adds a perturbation to the structure is not really well-defined for it takes a list of perturbations to add to the structure. The function is assumed to add multiple sentences with type insertion, multiple with type deletion etc. to the perturbation structure. The functions char-insertion, char-deletion, char-substitution and char-transposition actually perturb one single word into a number of different new words.

Character insertion

The first perturbation function is the function called char-insertion.

```
%% insertion error, characters must be deleted
char-insertion(word) {
   for (j=0 to length(word); j++) {
      p = word[0..j-1]+word[j+1..length(word)];
      if spellcheck(p) then
         parselist.add(sentence[0..i-1]+p+
                                      sentence[i+1..length(sentence)]);
   }
   return parselist;
}
```

The function name is a bit misleading; insertion in this case means that it tries to detect an insertion error. What it actually does is exactly the opposite. It takes a single word as input and then, iterating over the length of the word, a single character is deleted and the newly created word is checked against some list of words. The latter is what the undefined function 'spellcheck' does. The spellchecking might be done by using a traditional spellchecker, another possibility is to check the word against some available lexicon.

Character deletion

The second character perturbation function is char-deletion.

```
%% deletion error, characters must be inserted
char-deletion(word) {
   for (j=0 to length(word); j++) {
      for (char=' ' to 'z'; char++) { %% this includes a space
         p = word[0..j]+char+word[j..length(word)];
         if spellcheck(p) then
            parselist.add(sentence[0..i-1]+p+
                                      sentence[i+1..length(sentence)]);
      }
   }
   return parselist;
}
```

As for the previous function, its name is misleading. This function inserts characters to try and correct a deletion error. Note that the list of characters to be inserted includes a whitespace character which means that possibly two new words are constructed, the spellcheck function needs to take this into account.

Character substitution

The third function, char-substitution, does not have a misleading name for a change. Like the previous functions, it iterates over the length of the input word, creating a new word in which one character is replaced with one from a list of characters including a whitespace. Again, the newly created word is spellchecked, taking into account the possibility of it consisting of two separate words.

```
char-substitution(word) {
   for (j=0 to length(word); j++) {
      for (char=' ' to 'z'; char++) {
         p = word[0..j-1]+char+word[j+1..length(word)];
         if spellcheck(p) then
            parselist.add(sentence[0..i-1]+p+
                                    sentence[i+1..length(sentence)]);
      }
   }
   return parselist;
}
```

Character transposition

The fourth and last character perturbation function is char-transposition. Every two consecutive characters are interchanged to create a new word.

```
char-transposition(word) {
   for (i=1 to length(word); i++) {
      p = word[0..j-2]+word[j]+word[j-1]+word[j+1..length(word)];
       if spellcheck(p) then
          parselist.add(sentence[0..i-1]+p+
                                    sentence[i+1..length(sentence)]);
   }
   return parselist;
}
```

The function starts its iteration at the second character instead of the first. This is because it interchanges the character with the preceding. As usual, the spellchecker makes sure that nonwords do not even get parsed.

When all character level perturbations are made, what is constructed is a list of perturbations (sentences), every single one having exactly one character level perturbation. All sentences together form every perturbation of the input sentence for which the newly formed word is actually in the lexicon. This list will still contain a lot of sentences that we do not want to parse all. The penalty that is assigned to every sentence, based on the perturbation type can be used to decide if the sentence should be parsed or not. Section 4.3 goes into more detail on these penalties.

4.2.2 Word level perturbations

For making character level perturbations we could just try and perturb every word in every kind of way, because we know beforehand that there will be a large number of perturbations that will not survive the inexpensive operation of spellchecking. Unfortunately, this is not the case for word level perturbations. At first glance, it seems that every word level perturbation has to be parsed to check if it is actually a good perturbation. But as I said before, parsing a sentence is a very time-expensive operation, so we only want to

parse perturbed sentences of which we have a considerable indication that they might be a good alternative to the original sentence. Otherwise, it is a waste of computing power and thus a waste of time.

So we need a way of determining if a word level perturbation has a chance of being parsed correctly without having to parse it. One way of doing this is by consulting a big corpus of sentences and using probabilistic techniques based on measures of n-gram probability. This has already been described in section 2.2.4. We assess a perturbation by seeing how likely the resulting n-grams are in a corpus of correct sentences. However, there are some problems with using a corpus of correct sentences. There is still a very large search space, since we need to start off by generating all possible word order perturbations, even if we do not need to parse them. What real teachers do is to remember the frequent mistakes. Certain mistakes are more typical than others so we want to look at the probability of a whole perturbation, rather than just the probability of the correct sentence. The rest of this section describes my proposal on how to do this.

A corpus of word level error corrections

To compute the probability of a perturbation, we need a corpus of word level error corrections. Such a corpus can easily be compiled using existing corrected students' essays. Those essays are corrected by teachers which means that both the incorrect sentence containing a typical error and the right correction of this error is available. The only thing that needs to be done is to put both the original sentence and the corrected version of the sentence in a table of corrections. I will refer to this table as a corpus of applications of rewriting rules or perturbations.

Making a table of correction events

The perturbation corpus now consists of perturbations involving complete sentences. But a perturbation involves only one or two words. Naturally we want to take into account the context of any given perturbation, so we should include the immediate flanking words on either side of the perturbation. However, aside from those flanking words, the rest of the words in the sentence can be discarded.

I created a function which compiles a list of rewriting rules into a format that can be used by the perturbation algorithm. I will call this function 'compile-corpus'. The pseudocode for this function is given below:

```
compile-corpus([(original,correction)]) {
   create-hash-table(original, [(correction, cnt)]);
   for every (original, correction) in [(original, correction)] do {
      store-in-hash(create-trigram-perturbation(original, correction));
      store-in-hash(create-bigram-perturbations(original, correction));
      store-in-hash(create-unigram-perturbation(original, correction));
   }
}
```

The input of the function is the complete corpus of sentence corrections, i.e. a list of tuples containing the original sentence and the corrected version of the sentence. For explanatory purposes I will assume a corpus containing only two rewriting rules. This example corpus is a list with two tuples, i.e. (("I like the nature very much" , "I like nature very much") , ("I saw a dog big" , "I saw a big dog")).

First the 'compile-corpus' function creates a hash-table, using the original input as index. Associated with the original input is a list of corrections together with their count. This hash-table can be compared to a cupboard with an infinite number of labeled drawers. Every drawer can be used to store information about the object on its label.

The function 'create-trigram-perturbation' returns a tuple of trigrams containing only the flanking words in the perturbation, which are surrounded by asterices. Essentially it strips off the leftmost and rightmost words that are equal in both sentences. If no word remains, a keyholder '*_GAP_*' is inserted; if more than one word remains, the words will be considered part of one correction. The words left and right of the flanking words are reinserted leaving two trigrams, each of which share the same left and right words. Taking the example as input, in the first iteration the function returns the tuple ("like *the* nature" , "like *_GAP_* nature") and in the second iteration the tuple ("a *dog big* _EOS_" , "a *big dog" _EOS_").

The keyholder '_EOS_' is used to denote the end of the sentence, a similar keyholder '_SOS_' is used to denote the start of a sentence.

The word(s) in between the asterices are considered to be the actual correction. This trigram perturbation tuple is then stored in the hash-table, using the original trigram as index. If the exact trigram perturbation already exists in the hash-table, its count is increased by one. Otherwise, a new tuple containing the corrected trigram and a count of one is stored in the hash-table. To get back on the example, if there already is a drawer labeled "like *the* nature" we use that one, otherwise we use an empty drawer and label it likewise. Then, we put the perturbation "like *_GAP_* nature" inside this drawer. The same process is repeated for the other trigram tuple.

The function 'create-bigram-perturbations' does almost the same for bigram tuples. In the implementation however, it takes the trigram perturbations to start with and first strips off the leftmost word. This bigram perturbation is stored in the hash-table in the same way the trigram perturbations was. Then the rightmost word is stripped off the trigram perturbation and stored in the hash-table. For the example corpus, both the tuples ("like *the*" , "like *_GAP_*") and ("*the* nature" , "*_GAP_* nature") are stored as well as the tuples ("a *dog big*" , "a *big dog*") and ("*dog big* _EOS_" , "*big dog* _EOS_").

The function 'create-unigram-perturbation' strips off both the leftmost and the rightmost word from the trigram perturbation and stores this unigram perturbation, which consists of only the word(s) inside the asterices, in the hash-table.

Generating candidate perturbations

Now that we have got a table of existing trigram perturbations, we can use this to create perturbations for the input to the system. The function that creates the word level perturbations looks similar to the function that created the character level perturbations:

```
make_word_perturbations(sentence) {
   [sentence] parselist;
   (sentence, type, penalty) perturbation;

   perturbation.add(word-insertion(sentence),
                              "word-insertion", word_insertion_penalty);
   perturbation.add(word-deletion(sentence),
                                "word-deletion", word_deletion_penalty);
   perturbation.add(word-substitution(sentence),
                        "word-substitution", word_substitution_penalty);
   perturbation.add(word-transposition(sentence),
                      "word-transposition", word_transposition_penalty);
   return perturbation;
}
```

The difference with the function that creates the candidate character perturbations is that the four functions that correspond to the four error types now take the whole sentence as an argument. Again, the functions that actually create the perturbations are the functions word-insertion, word-deletion, word-substitution and word-transposition. I will explain those four functions below.

Word insertion

```
word-insertion(sentence) {
   for (i=0 to length(sentence); i++) {
      [(correction, cnt)] =
         get-from-hash("*" + sentence[i] + "* " + sentence[i+1]);
      for every (correction,cnt) in [(correction,cnt)] do {
         if correction = "*" + sentence[i] + "* _GAP_" then
            parselist.add(sentence[0..i] + " " +
                                       sentence[i+2..length(sentence)]);
```

```
         if correction = "_GAP_ *" + sentence[i] + "*" then
            parselist.add(sentence[0..i-1] + " " +
                                       sentence[i+1..length(sentence)]);
      }
   }
   return parselist;
}
```

Iterating over all the words in the sentence, every sequence of two words is looked up in the hash-table. If there is a record in the hash-table for which this sequence of two words can be rewritten to a sequence of two words containing a gap, then we have got evidence for the perturbation to be correct. In this case, the rwriting is applied to the original sentence and the result is stored in the list of sentences to be parsed. At the end, this list is returned.

Word deletion

```
word-deletion(sentence) {
   for (i=0 to length(sentence); i++) {
      [(correction, cnt)] =
         get-from-hash("*" + sentence[i] + "* _GAP_");
      if exists [(correction, cnt)] then
         for every (correction, cnt) in [(correction, cnt)] do
            parselist.add(sentence[0..i-1] + correction +
                                       sentence[i..length(sentence)]);
      [(correction, cnt)] =
         get-from-hash("_GAP_ *" + sentence[i] + "*");
      if exists [(correction, cnt)] then
         for every (correction, cnt) in [(correction, cnt)] do
            parselist.add(sentence[0..i] + correction +
                                       sentence[i+1..length(sentence)]);
   }
   return parselist;
}
```

Again iterating over the words in the sentence, this function tries to find evidence in the corpus for a deletion. It first looks in the hash-table for an occurrence of the word followed by the keyholder "_GAP_". If such an index is found, every possible correction is applied to the original sentence and stored in the parse list. The same thing happens for indices of the word preceded by this keyholder.

Word substitution

```
word-substitution(sentence) {
   for (i=0 to length(sentence); i++) {
      [(correction, cnt)] = get-from-hash("*" + sentence[i] + "*");
      if exists [(correction, cnt)] then
         for every (correction, cnt) in [(correction, cnt)] do
            parselist.add(sentence[0..i-1] + correction +
                                       sentence[i+1..length(sentence)]);
   }
   return parselist;
}
```

The two previous functions looked up a bigram in the corpus. For detecting a word substitution, the existence of one single word in the corpus is assumed to be enough evidence. Again, every correction that can be made using this word is applied to the original sentence.

Word transposition

```
word-transposition(sentence) {
   for (i=1 to length(sentence); i++) {
      [(correction, cnt)] = get-from-hash("*"+sentence[i-1..i]+"*");
      if exists [(correction, cnt)] then
         for every (correction, cnt) in [(correction, cnt)] do
           parselist.add(sentence[0..i-1] + correction +
                                    sentence[i+1..length(sentence)]);
   }
   return parselist;
}
```

The word transposition function is the most tricky. In the creation of the error rule corpus, the flanking words were surrounded by asterices. The two words that are looked up in the corpus are part of a transposition and thus are both flanking words. We know that the corpus only consists of trigrams containing one of the four error types. Transposition is the only type for which two words are involved in the correction. Therefore, we do not have to check if the correction really transposes the two flanking words.

The preceding four functions create all possible word level perturbations. The number of created perturbations will be limited because only perturbations for which evidence was found in the correction corpus are created. All those perturbations will eventually be sent to the parser. But the reason for creating the correction corpus in the first place was to give a probability to a perturbation. To compute a perturbation rule probability, the technique of n-gram corpus lookup has to be transformed into a technique of n-gram perturbation rule lookup. In chapter 5 I will make an analogy between the well-known statistical natural language processing technique of n-gram corpus lookup and my own n-gram perturbation corpus lookup technique.

4.3 Perturbation penalties

In the previous section I mentioned a penalty to be assigned to certain perturbation operations. In this section I will go into more detail on how to determine the values of these penalties and how to combine them into one overall score. We need an overall score to be able to decide whether we want to parse a perturbed sentence. If the overall score is above a certain threshold value, there is assumed to be enough evidence that the newly created sentence might be correct and that it is not a waste of time trying to parse it.

For assigning a penalty to a perturbation, the distinction between the character and word level is very important. While errors at the character level are usually spelling errors, errors at the word level are more likely to be caused by an incorrect application of grammar rules. The penalties for word and character level are therefore assigned independently using different measures.

4.3.1 Character level perturbations

In chapter 3 I made a classification of perturbation types. On two levels, there are four different perturbation types. But as I explain in section 3.4, on the character level there are multiple causes for every one of the four perturbation types, some of which are more likely than others. This gives ground to the assignment of different penalties to the different errors. Apart from the arbitrary character insertion, deletion, substitution and transposition errors, a few special cases have been recognized. These are:

1. Insertion typo
2. Doubling insertion
3. Doubling deletion
4. Substitution typo

5. Phonological substitution

These error types can be categorized into typos (1 and 4), doubling errors (2 and 3) and a phonological error (5). For all of these categories we can say that if a perturbation constructs a valid word and can be categorized in one of them, this perturbation is more likely than an arbitrary one. Therefore, perturbations in those three subcategories must be assigned a lower penalty. Furthermore, transposition errors must be treated separately, for they are less likely to construct valid words.

I define the type specific perturbation penalty given to a character level perturbation to be an integer between 0 and 10. The penalties I assign to the different perturbation types are given in table 4.1. The parameters given in the second row of the table are lisp global variables. This means that they are easy to change and play around with. I derived these values by experimenting a little with the system using

type	*parameter name*	*penalty*
typo	*penalty-close-keys*	1
phonological	*penalty-close-sound*	2
doubling	*penalty-doubling*	3
transposition	*penalty-swap*	6
other	*penalty-unknown*	9

Table 4.1: Perturbation type penalties

different values for the parameters. I came up with the given values, but some more thorough testing should be done to improve the system's performance.

There is a second source of information to rely on for the evaluation of a perturbation. Testing the algorithm I found that the system came up with much more possible perturbations for shorter words. For example, the perturbations the system came up with for the word *teh*, which is quite obviously a misspelling of *the*, were *the*, *tea*, *ted*, *ten*, *t eh*, *tech* and *eh*. It gets even worse when the system starts perturbing the correct word *the*. The (10!) possible perturbations of the word *the* are *she*, *tie*, *thx*, *t he*, *thed*, *them*, *then*, *thes*, *they* and *he*. We most certainly do not want the system to parse all of those ten perturbations every time the word *the* is found in a sentence. Fortunately, longer words tend to have many fewer possible perturbations. The word length is a good measure to take into account.

Now, we have got two measures for the likelihood of a perturbation. We need to combine those two measures into one score for the character level perturbation. I actually want to have a weighted average of the two scores, but before I can do so, I need to normalize them. I normalize both scores to have a value between -1 and 1. The more positive the value, the more likely the perturbation is. The normalizing functions I used are:

$$\text{norm-penalty} = \frac{type\ penalty - 5}{-5} \tag{4.1}$$

$$\text{norm-length} = \sqrt{word\ length} - \sqrt{3} \tag{4.2}$$

The second function is set to have a maximum of 1. Again, these functions are derived from some initial testing, some more thorough experimenting could be done to produce better results.

The weighted average is computed by means of a global variable called **char-factors**. This variable holds a list of numbers, each corresponding to one measure. In this case, there are two factors; one for the type specific penalty and one for the word length. The weighted sum can now be computed using the following equation:

$$\frac{\sum_{m=0}^{n} f_m p_m}{n} \tag{4.3}$$

In this equation, the n stands for the number of different measures (2 in our case), f stands for the factor of measure m and p stands for the normalized score. After computing the final character level score, the algorithm creates a short list of perturbations that have a positive score. Those are the only perturbations that will be parsed.

4.3.2 Word level perturbations

On a word level, scoring based on perturbation type is not much use. A score for the word level perturbations can be assigned using the *probability* of the perturbation. In chapter 5 I will explain in detail how a probability for a perturbation is computed. For now, assume we can compute this probability. The nice thing about the algorithm that creates the word level perturbations is that it consults the corpus of applications of rewriting rules to create the perturbations in the first place. The probability of a perturbation always lies between 0 and 1. There is no need to eliminate certain perturbations, so the score given to a word level perturbation is equal to its probability.

4.3.3 Overall perturbation score

I just explained how both on the character level as on the word level, a final score is assigned which is always a number between 0 and 1. Although we can not really compare the likelihood of a character perturbation to the likelihood of a word level perturbation, we have given both of them a score between 0 and 1. By tweaking the parameters for the character level scores, the system can be optimized so that a right balance between the word level probability and the character level scores is achieved.

Chapter 5

The backoff algorithm

N-gram techniques have been used in the implementation of error correction systems, mainly because they are not very expensive in terms of execution time. The only thing they rely on is a sufficiently large corpus of sentences. Because of this low cost aspect, n-gram techniques are also useful for the algorithm that creates candidate perturbations. Although the traditional n-gram technique cannot be used directly, this technique can be transformed into a technique that can be used by the perturbation algorithm. Jurafsky and Martin [6] give a good description of some of the techniques that are used for predicting what the next word in a sentence could be. In this chapter, I will explain how these algorithms work and how they can be used by the perturbation algorithm.

5.1 N-gram probabilities

5.1.1 Simple n-grams

If we have an unfinished sentence and we want to know what the next word in the sentence might be, we can use a corpus to try and predict the next word. The only thing that needs to be done is to take all the sentences in the corpus that start with the unfinished sentence. We then collect all the single words that follow this sentence. Words in this collection with a higher count are more likely to be the next word in the sentence than words with a lower count. Even more, every word in this collection can be assigned a probability based on its count. The probability of a word following the sentence can be computed by taking the collection of all words following the sentence and dividing the count of the particular word by the total size of the collection. The probabilities of all words in the collection will sum to one, so this is indeed a well-defined probability.

For this to work, we need an infeasibly large corpus. The longer the unfinished sentence, the lower the chance of finding an exact match of the sentence in the corpus and the worse the lookup method will work. Now, does the next word in a sentence rely on the complete unfinished sentence? Yes, it does, but the most influential are the last n words in the sentence. In other words, the prediction of the next word in a sentence can be simplified by predicting the word that follows the last n-gram in the sentence. Talking about probabilities, the probability of the next word in a sentence can be estimated by looking at the probability of seeing the last word, given the fact that we have just seen the exact sequence of the n previous words. For example, take the sentence *I really like his dog*. If we consider each word occurring in its exact location as an independent event, the probability of the word *dog* following the exact sentence *I really like his* can be approximated by just looking at the trigram *like his dog*. This probability can be computed by dividing the counts of the word *dog* by the total number of words that can follow *like his*.

$$P(I\ really\ like\ his\ dog) \simeq P(like\ his\ dog) = \frac{c(like\ his\ dog)}{c(like\ his)} \tag{5.1}$$

5.1.2 Perturbation n-grams

In the case of perturbations, what we want to do is not to compute the probability of a certain word occurring in a text, but we want to know what the probability is of a certain perturbation. Instead of looking up a word in a grammatically correct corpus of words, we need to look up a rewriting rule in a corpus of applications of rewriting rules. So, if we want to know what the probability is of a perturbation *(I really like him dog → I really like his dog)*, we can compute this probability by looking at the trigram perturbation *(like him dog → like his dog)*.

$$P(I\ really\ like\ him\ dog \rightarrow I\ really\ like\ his\ dog) \simeq P(like\ him\ dog \rightarrow like\ his\ dog) \tag{5.2}$$

The probability of an n-gram perturbation can be computed in a similar way as we computed the probability of a normal n-gram. We count the number of times the particular perturbation occurs in the corpus and divide this by the total number of perturbations that can be made from the same left-hand side of the rule.

$$P(like\ him\ dog \rightarrow like\ his\ dog) = \frac{c(like\ him\ dog \rightarrow like\ his\ dog)}{c(LHS(like\ him\ dog))} \tag{5.3}$$

This is again a well-defined probability. If, for example, the only possible way of correcting *like him dog* would be to rewrite it to *like his dog*, then there would be a number of rewriting rules containing this perturbation. Exactly the same number of rewriting rules would have *like him dog* in their left-hand side. The probability will be 1, as expected, as there is no other way of rewriting *like him dog*. If there are other ways of rewriting it, only the denominator of the fraction will increase and the total probability will decrease.

5.2 Discounting

Both the n-gram lookup technique as well as the derived perturbation rule lookup technique work as long as there is a substantial number of n-grams or perturbation rules in the corpus. But if a certain n-gram or perturbation is not contained in the corpus, its probability will be zero. Some of these possibilities might still be good, so we will give the zero-count events a slightly higher count. This technique is called discounting and several algorithms have been developed to do this. I will explain how the discounting algorithm described by Witten and Bell works and how to apply this algorithm to the perturbation counts.

5.2.1 Witten-Bell discounting

The key concept in the algorithm is a recurring concept in statistical language processing. This concept is to use the count of things we have already seen to estimate the count of things we have not seen. In other words, I am going to use the total number of different n-grams to estimate the probability of the n-grams that are not contained in the corpus. The goal is to assign a probability to n-grams that are not in the corpus. Let's take the example of *like his dog* and assume that the count of *like his dog* in the corpus is zero. The probability I am going to assign to this trigram is the probability that there is one instead of no occurrence of *like his dog*. The count of all the non-occurring trigrams that start with *like his* is defined to be the number of different trigrams in the corpus that start with this bigram. I will call this *T*, the number of (existing) trigram types. Now, I want to assign a probability to each of these trigrams, so I try and estimate the sum of their probabilities. This sum is called the 'probability mass' of the trigrams. We estimate the total probability mass by dividing T by the number of types plus *N*, the total possible number of trigrams that start with *like his*. N is actually equal to the dictionary size because every word in the dictionary could possibly follow the bigram.

$$\sum_{w_n : c(like\ his\ [w_n])=0} P(like\ his\ [w_n]) = \frac{T}{T+N} \tag{5.4}$$

If I define *Z* to be the total number of trigrams starting with *like his* that are not contained in the corpus, the probability to assign to *like his dog* can be computed by dividing this total probability mass by Z.

$$P(like\ his\ dog) = \frac{T}{Z(T+N)} \tag{5.5}$$

The only problem here is that I introduced extra probability mass by assigning probabilities to the zero-count trigrams. This mass comes from the non zero-count trigrams. To account for this, the probability of a non zero-count trigram is discounted as follows:

$$P(trigram) = \frac{c(trigram)}{T+N} \tag{5.6}$$

The complete discounting model, giving the new counts as c^*, can be represented as follows:

$$c^*(trigram) = \begin{cases} \frac{T}{Z}\frac{N}{T+N}, & \text{if } c(trigram) = 0 \\ c(trigram)\frac{N}{T+N}, & \text{if } c(trigram) > 0 \end{cases} \tag{5.7}$$

5.2.2 Perturbation discounting

In the case of perturbation rules, the definitions of T, N and Z are slightly different. For example, take the perturbation *(like him dog → like his dog)*. Assume that this exact perturbation is not in the corpus but there are other perturbations of *like him dog*. I now estimate the probability of the first perturbation by looking at all the possible perturbations that can be constructed from *like him dog*. I need to redefine T to be the total number of different perturbations in the corpus with *like him dog* in their left-hand side. N is again the number of different words in the dictionary, Z is the total number of possible perturbations that are not contained in the dictionary. The discounted perturbation rule counts can be computed using formula (5.7).

5.3 Backoff

Discounting is a good solution for assigning probability mass to perturbations that are not contained in the corpus, yet there is another method of solving the problem. This is the backoff method I discussed in section 2.2.5. Katz [7] introduced an n-gram backoff model for backing off to lower order n-grams. Analogous to the construction of the perturbation discounting method, I will explain the original backoff model and the necessary changes for applying the model to perturbation rules.

5.3.1 Katz' backoff model

In the backoff model, if the number of n-gram counts in a corpus is zero, instead of giving this n-gram a probability of zero, the counts of the (n-1)-gram in the corpus is used. The trigram version of this mechanism might be represented as follows:

$$\widehat{P}(dog|like\ his) = \begin{cases} \widetilde{P}(dog|like\ his), & \text{if } c(like\ his\ dog) > 0 \\ \alpha_1\widetilde{P}(dog|his), & \text{if } c(like\ his\ dog) = 0 \\ & \text{and } c(his\ dog) > 0 \\ \alpha_2\widetilde{P}(dog), & \text{otherwise.} \end{cases} \tag{5.8}$$

The α-values and the $\widetilde{P}$ are needed because without them, the overall probability $\widehat{P}$ would not be a true probability. A true probability always sums to one when you take the sum of the probabilities of all bigrams that can precede the given word. We want the overall probability to be a well-defined probability, so the total probability mass of the backoff model needs to sum to one. We know that $\sum_{i,j} P(w_n|w_iw_j) = 1$, so in order to back off to a bigram model for the given word, probability mass has to be distributed from the trigram to the bigram model. Witten-Bell discounting is used to determine the amount of probability mass to distribute from the trigram model to the lower order models. The $\widetilde{P}$ in this model stands for a discounted probability and can be computed as follows:

$$\widetilde{P}(dog|like\ his) = \frac{c^*(like\ his\ dog)}{c(like\ his)} \tag{5.9}$$

The $alpha$-values are actually functions which distribute the left-over probability mass of a (n+1)-order model over the current n-gram model. To apply this to our trigram model, the first thing we need to know

is the total probability mass that is left over from the trigram model. The trigram backoff model only backs off to a lower order model if the count of the trigram is zero. So the total probability mass that stays in the trigram model can be computed by looking at the total probability of all the non-zero count trigrams that start with the same bigram. This sum is subtracted from the total probability mass of the trigram model which is naturally one. This is represented by the β-function:

$$\beta(w_{n-2}^{n-1}) = 1 - \sum_{w_n : c(w_{n-2}^{n}) > 0} \widetilde{P}(w_n | w_{n-2}^{n-1}) \tag{5.10}$$

To determine how much probability mass is distributed to the bigram under consideration, the total mass needs to be normalized by the total probability of all non-zero count bigrams that begin some trigram. This is represented by the γ-function:

$$\gamma(w_{n-2}^{n-1}) = 1 - \sum_{w_n : c(w_{n-2}^{n}) > 0} \widetilde{P}(w_n | w_{n-1}) \tag{5.11}$$

Now that we know the β- and γ-functions, we can do the computation of the α-functions for our example. The α_1 for the trigram *like his dog* is a function of *like his*:

$$\alpha_1 = \alpha(like\ his) = \frac{\beta(like\ his)}{\gamma(like\ his)} = \frac{1 - \sum_{w_n : c(like\ his\ [w_n]) > 0} \widetilde{P}(w_n | like\ his)}{1 - \sum_{w_n : c(like\ his\ [w_n]) > 0} \widetilde{P}(w_n | his)} \tag{5.12}$$

α_2 is a function of *his*:

$$\alpha_2 = \alpha(his) = \frac{\beta(his)}{\gamma(his)} = \frac{1 - \sum_{w_n : c(his\ [w_n]) > 0} \widetilde{P}(w_n | his)}{1 - \sum_{w_n : c(his\ [w_n]) > 0} \widetilde{P}(w_n)} \tag{5.13}$$

5.3.2 Perturbation backoff model

For the error rule corpus the backoff model is very similar. Keeping in mind that (5.1) is analogous to (5.3), the backoff model looks like this for the example perturbation:

$\widehat{P}(like\ him\ dog \rightarrow like\ his\ dog) =$

$$\begin{cases} \widetilde{P}(like\ him\ dog \rightarrow like\ his\ dog), & \text{if } c(like\ him\ dog \rightarrow like\ his\ dog) > 0 \\ \frac{1}{2}\alpha_{tri-left}\widetilde{P}(like\ him \rightarrow like\ his) & \\ +\frac{1}{2}\alpha_{tri-right}\widetilde{P}(him\ dog \rightarrow his\ dog), & \text{if } c(like\ him\ dog \rightarrow like\ his\ dog) = 0 \\ \alpha_{bi}\widetilde{P}(him \rightarrow his), & \text{otherwise.} \end{cases} \tag{5.14}$$

$$\widetilde{P}(like\ him\ dog \rightarrow like\ his\ dog) = \frac{c^*(like\ him\ dog \rightarrow like\ his\ dog)}{c(like\ him\ dog)} \tag{5.15}$$

The biggest difference is found in the backoff to the bigram model. When looking up words in a corpus, the probability of a trigram is dependent on the probability of the bigram that starts the trigram. However, the probability of a trigram rewriting rule is dependent on both the probability of the two leftmost words as well as the probability of the two rightmost words. Both of these probabilities have to be taken into account and the probability mass is ditributed equally over the two bigram rules. Therefore, I average the probabilities by adding them and dividing them by 2. If none of the bigram perturbations has a non-zero count, we backoff to the unigram model.

When the counts of a trigram perturbation are zero, the total probability mass that needs to be be distributed from the trigram model to the bigram model is given by the function β_{tri}:

$$\beta_{tri}(like\ him\ dog) = 1 - \sum_{w_n : c(like\ him\ dog \rightarrow like\ [w_n]\ dog) > 0} \widetilde{P}(like\ him\ dog \rightarrow like\ [w_n]\ dog) \tag{5.16}$$

The γ-function needed for normalizing depends on the fact if we are backing off to the bigram on the left-hand side or the right-hand side of the trigram:

$$\gamma_{tri-left}(like\ him\ dog) = 1 - \sum_{w_n : c(like\ him\ dog \rightarrow like\ [w_n]\ dog) > 0} \widetilde{P}(like\ him \rightarrow like\ [w_n]) \quad (5.17)$$

$$\gamma_{tri-right}(like\ him\ dog) = 1 - \sum_{w_n : c(like\ him\ dog \rightarrow like\ [w_n]\ dog) > 0} \widetilde{P}(him\ dog \rightarrow [w_n]\ dog) \quad (5.18)$$

The α-functions are given below:

$$\alpha_{tri-left}(like\ him\ dog) = \frac{\beta_{tri}(like\ him\ dog)}{\gamma_{tri-left}(like\ him\ dog)} \quad (5.19)$$

$$\alpha_{tri-right}(like\ him\ dog) = \frac{\beta_{tri}(like\ him\ dog)}{\gamma_{tri-right}(like\ him\ dog)} \quad (5.20)$$

For backing off from bigram to unigram, things are a little bit different. The β-function gives the left-over probability mass from the bigram model. The left-over mass comes from both the left of the unigram as well as the right of the unigram.

$$\beta_{bi}(like\ him\ dog) = \beta_{bi-left}(like\ him\ dog) + \beta_{bi-right}(like\ him\ dog) \quad (5.21)$$

$$\beta_{bi-left}(like\ him\ dog) = 1 - \sum_{w_n : c(like\ him \rightarrow like\ [w_n]) > 0} \widetilde{P}(like\ him \rightarrow like\ [w_n]) \quad (5.22)$$

$$\beta_{bi-right}(like\ him\ dog) = 1 - \sum_{w_n : c(him\ dog \rightarrow [w_n]\ dog) > 0} \widetilde{P}(him\ dog \rightarrow [w_n]\ dog) \quad (5.23)$$

The γ-function distributes this mass over all the unigrams:

$$\gamma_{bi}(like\ him\ dog) = 1 - \sum_{w_n : (c(like\ him \rightarrow like\ [w_n]) + c(him\ dog \rightarrow [w_n]\ dog)) > 0} \widetilde{P}(him \rightarrow [w_n])) \quad (5.24)$$

The α_{bi}-function simply divides the β-function by the γ-function:

$$\alpha_{bi}(like\ him\ dog) = \frac{\beta_{bi}(like\ him\ dog)}{\gamma_{bi}(like\ him\ dog)} \quad (5.25)$$

5.4 Summary

We now have an algorithm for estimating the probability of any arbitrary word level perturbation, using a corpus of trigram correction events. The algorithm works as follows: First, we look up the trigram perturbation in the corpus. If the exact trigram perturbation is found, equation (5.15) is used to compute the probability of the perturbation. If the exact trigram perturbation is not found in the corpus, the algorithm backs off to a bigram model instead of a trigram model. A bigram perturbation can be constructed both from the two leftmost words and from the two rightmost words. If a bigram perturbation is found in the corpus, equation (5.15) is used to compute its probability. We multiply this probability by the α-function to account for the fact that we back off to a lower order model. If even the bigram perturbation can not be found in the corpus, we back off to a unigram model. The same equation is used to compute the unigram perturbation probability and this probability is multiplied by another α-function.

Chapter 6

Results

In this chapter, I will dicuss the results of inserting the perturbation module into Te Kaitito. In section 6.1, I will show the performance of the functions that create the perturbations. Section 6.2 gives the results of the integration of the module into Te Kaitito.

Unless otherwise specified, the values for the global variables in these results are as shown in appendix B.

6.1 Perturbation functions

For demonstrating how the perturbation functions work, I use the text-only lisp interface to the system. Using this interface I can directly call the functions that create the perturbations. These functions are 'create-char-perturbations' for creating the character level perturbations and 'create-word-perturbations' for creating the word level perturbations. Both of the functions return a list of possible (likely) perturbations, together with the exact type as defined in chapter 3 and the score associated with the perturbation.

6.1.1 Character level perturbations

To demonstrate the character level perturbations, I will first give an example that shows the correct functioning of the scoring system. Then, by way of an independent test, I will use the examples of perturbations given in table 3.1. I will create all likely perturbations for these examples; because the algorithm was not designed to process these examples, they count as unseen events and are a good means of evaluating the system.

For the examples in this section the value of the global variable *char-factors* is important. This variable holds the weighing factors for the type specific penalty and the word length. In creating the examples, I decided to rely equally on both factors, i.e. I used the default value $(1, 1)$ for this variable. Another set of variables that play a role are the type specific penalties themselves. The values given to those penalties are the default values given in appendix B and are also shown in table 4.1.

Insertion typo

The detection of an insertion typo involves comparing two adjacent letters in the word to see if the keys are close to each other on the keyboard. If so, an insertion typo is likely to have occurred. For testing I used a very short list of close key characters. The character pairs I defined to be close on the keyboard are: (t,y) (t,r) (h,j) and (s,a). The first example is the perturbation of the word *sdorted*.

```
TK> (create-char-perturbations "sdorted")
(("shorted" CHARACTER-SUBSTITUTION 0.056850243)
 ("sported" CHARACTER-SUBSTITUTION 0.056850243)
 ("sorted" INSERTION-TYPO 0.75871956))
```

There are three possible corrections of this nonword, two of which are generated by a general character substitution. Since the letters *s* and *d* are close on the keyboard, the insertion typo gets a much higher score.

The initial example involves the word *party*.

```
TK> (create-char-perturbations "patrty")
(("party" INSERTION-TYPO 0.65200865)
 ("patty" INSERTION-TYPO 0.65200865))
```

In this case, both the (first) *t* and the *r* have been chosen as candidates for removal. Apparently, the word *patty* is included in the lexicon. The perturbations are of the same length and type, therefore the score assigned to them is exactly the same. The score is somewhat lower than in the previous example because the word *party* has only five characters.

Doubling insertion error

For the detection of a doubling insertion error, two adjacent equal characters have to be detected. This should work for any possible character. My first example is the word *inittial*.

```
TK> (create-char-perturbations "inittial")
(("initial" DOUBLING-INSERTION 0.6568502))
```

This is the result we were looking for. Although the word is reasonably long, the score is low compared to the previous ones. This is because the doubling penalty is quite high, compared to the other type specific penalties.

The initial example involves the word *very*. This time, something interesting happens.

```
TK> (create-char-perturbations "verry")
(("ferry" PHONOLOGICAL-SUBSTITUTION 0.5520086)
 ("very" DOUBLING-INSERTION 0.3339746))
```

The example should show that the doubling insertion is detected by the system. Indeed, the system detects the doubling of the letters r, but the doubling insertion is not the perturbation with the highest score. Because of the relatively high penalty assigned to a doubling insertion as well as the fact that the created word only has four characters instead of five, the phonological substitution which creates the word *ferry* has a much higher score. The parsing process will make sure that the correct perturbation is selected, because *ferry* is a noun whereas *very* is an adverb. Only one of these will be grammatically correct in the context of a sentence.

What makes this example particularly interesting is that when you think about it, without taking the word into any context *ferry* is actually a more likely correction of the word *verry* than the one we were looking for.

Character insertion

The hardest to detect is the case in which there are no clues as to what caused the error in the first place. The next example shows that the system is still capable of detecting an arbitrary insertion.

```
TK> (create-char-perturbations "compjuter")
(("computer" CHARACTER-INSERTION 0.04818816))
```

As you can see, the score assigned to this perturbation is hardly enough for the system to even consider parsing it. But to a human, it is very obvious that an insertion error has been made. We could lower the penalty for an arbitrary insertion error or lower the acceptable word length, but the danger in this is that we also allow very unlikely perturbations to take place. A tradeoff between lowering penalties and allowing strange perturbations has to be made.

Doubling deletion error

A doubling deletion is easy to detect. We can just try to double every character in the word. The following example shows a common case.

```
TK> (create-char-perturbations "runing")
(("ruining" CHARACTER-DELETION 0.056850243)
 ("running" DOUBLING-DELETION 0.6568502)
 ("runsing" CHARACTER-DELETION 0.056850243)
 ("runings" CHARACTER-DELETION 0.056850243))
```

The example in chapter 3 works just as well.

```
TK> (create-char-perturbations "blury")
(("blurry" DOUBLING-DELETION 0.5587195))
```

Character deletion

This general deletion error is again the hardest case. There is no way of telling if a deletion is caused by a typo or not. The following example shows that it still works.

```
TK> (create-char-perturbations "telecope")
(("telescope" CHARACTER-DELETION 0.1))
```

Unfortunately, even though the word is very long, the score is still as low as 0.1. This is because the type specific penalty is quite high and weighs just as much as the word length.

The original example does not do very well.

```
TK> (create-char-perturbations "paty")
(("patty" DOUBLING-DELETION 0.45200863)
 ("pay" INSERTION-TYPO 0.4)
 ("pat" INSERTION-TYPO 0.4))
```

The deletion typo given in table 3.1 is a terrible example. There are three other perturbations that are much more likely than the one we were looking for. The sought after perturbation did not even survive the selection process. The most important reason for this is already given in chapter 3; there is no way of determining if a character deletion error is actually a typo or not. The penalty assigned to a deletion error is therefore very high. Even shorter words with a more likely perturbation type score much better.

Keyboard substitution typo

Usually, (keyboard) substitution typos construct completely nonsense words and the only way of perturbing such a nonword is by reconstructing the correct word. The following example shows this:

```
TK> (create-char-perturbations "lighy")
(("light" SUBSTITUTION-TYPO 0.65200865))
```

The initial example does equally well.

```
TK> (create-char-perturbations "psrty")
(("party" SUBSTITUTION-TYPO 0.65200865))
```

This error is typical in computer aided applications where a keyboard is used to get input from the user.

Phonological substitution error

This kind of error depends on a correct list of pairs of characters that are close in sound. For the examples, I used the following pairs: (d,t) (i,y) (v,f). The first example is *breat*.

```
TK> (create-char-perturbations "breat")
(("bread" PHONOLOGICAL-SUBSTITUTION 0.5520086))
```

The second example is just as good.

```
TK> (create-char-perturbations "parti")
(("party" PHONOLOGICAL-SUBSTITUTION 0.5520086))
```

Character substitution error

A little harder to detect than a special case of a substitution error, but still easier to detect than a character insertion or deletion is a character substitution error. The example I use is *splerdid*.

```
TK> (create-char-perturbations "splerdid")
(("splendid" CHARACTER-SUBSTITUTION 0.1))
```

Again, this score is very low, because the type specific penalty is so high.

Character transposition error

The last character level example is transposition. Transposition errors receive a lower penalty than other ordinary errors. The following example shows that this is a good thing.

```
TK> (create-char-perturbations "bakring")
(("barking" CHARACTER-TRANSPOSITION 0.35685024)
 ("barring" CHARACTER-SUBSTITUTION 0.056850243)
 ("bakeing" CHARACTER-SUBSTITUTION 0.056850243))
```

The correct perturbation, *barking*, is indeed the best option. The score is not very high but it is high enough to favour the correct substitution instead of one of the other two.

The example given in table 3.1 was the following:

```
TK> (create-char-perturbations "patry")
(("party" CHARACTER-TRANSPOSITION 0.15200862)
 ("patty" SUBSTITUTION-TYPO 0.65200865))
```

This example is one of the few that does not give a very satisfying result. Apart from the (correct) transposition, there is also the (higher) possibility of a substitution typo to have taken place. Substitution still has quite a high penalty and some tweaking of the penalties might help to give the correct result for this example. On the other hand, this is only one example and it might just be an unlucky one.

6.1.2 Word level perturbations

To demonstrate the performance of the word level perturbations, I need an example corpus containing at least one example of every perturbation type. The English error corpus I used for creating the examples in this section is given in figure 6.1. For every perturbation type, I will give some examples and explain the results.

word insertion errors
("I like the nature" → "I like nature")
("He plays the football" → "He plays football")
("I play the tennis" → "I play tennis")
deletion errors
("Sarah chases cat" → "Sarah chases the cat")
("John likes cat" → "John likes a cat")
word substitution errors
("I like him dog" → "I like his dog")
("I like she dog" → "I like her dog")
("I like their dog" → "I like her dog")
("We like hers dog" → "We like her dog")
word transposition errors
("I saw a dog big" → "I saw a big dog")

Figure 6.1: English test corpus of error rules

Word insertion error

In the first example I simply use the exact perturbation on the first line of the error corpus.

```
TK> (create-word-perturbations "Do you like the nature?")
(("do you like nature" WORD-INSERTION 0.5))
```

The system comes up with the correct perturbation. The probability for this perturbation is 0.5. This is due to the discounting algorithm. Since there is only one rule in which *like *the* nature* is mentioned, half of the probability mass is set aside for unseen perturbations of this sentence.

If I reinsert the first rule twice, so that the rule is mentioned three times in the corpus, we immediately see result.

```
TK> (create-word-perturbations "Do you like the nature?")
(("do you like nature" WORD-INSERTION 0.75))
```

Since the trigram *like *the* nature* is now mentioned twice, much less probability mass needs to be assigned to unseen events. As we would expect, this increases the probability of the rule.

It becomes even more interesting when we combine information from different rules. I first create the perturbations for the sentence *I watch the tennis*:

```
TK> (create-word-perturbations "I watch the tennis")
(("i watch tennis" WORD-INSERTION 0.25))
```

The system comes up with the right correction, but the proability is not as high as before. This is because the backoff system had to back off to a bigram model to find the perturbation. But now look at the next example:

```
TK> (create-word-perturbations "I like the tennis")
(("i like tennis" WORD-INSERTION 0.625))
```

This probability is much higher, because even though the backoff model backed off to a bigram model, the number of rules in which this bigram perturbation is found is very high (Remember that the first rule is still mentioned three times). Information from different perturbation rules is combined to give a better result. Notice that the probability is still lower than the probability of *like the nature*. Because of the backoff, the probability of a backed off rule can never be as high as the probability of an exact match of the trigram rule containing the bigram.

Word deletion error

The first word deletion error example I give is the simple case of the first word deletion error rule in the corpus.

```
TK> (create-word-perturbations "Sarah chases cat")
(("sarah chases the cat" WORD-DELETION 0.5)
 ("sarah chases a cat" WORD-DELETION 0.125))
```

The first (and highest scoring) option is again the one that is found exactly in the corpus. Also, we already see the backoff mechanism working. According to the rule *(John likes cat → John likes a cat)*, the bigram **_GAP_* cat* can be rewritten to **a* cat*. This fact is taken into account, but as there are is yet another way of rewriting **_GAP_* cat*, this probability is quite low.

You would expect the second probability to increase if there would be more evidence of the rule *(*_GAP_* cat → *a* cat)*. To show this happening, I insert two extra rules in the corpus, i.e. *(John sees cat → John sees a cat)* and *(John holds cat → John holds a cat)*. This is what happens to the probability:

```
TK> (create-word-perturbations "Sarah chases cat")
(("sarah chases the cat" WORD-DELETION 0.5)
 ("sarah chases a cat" WORD-DELETION 0.25))
```

As expected, the probability of the second perturbation increases, but because of the backoff to the bigram model, the probability will never be as high as the first one.

Word substitution error

A word substitution is not much different from the other word level perturbations. The first substitution rule in the error corpus gives the expected result:

```
TK> (create-word-perturbations "I like him dog")
(("i like his dog" WORD-SUBSTITUTION 0.5))
```

To show that multiple rules having the same right-hand side do not influence the probability of either one of them, the corpus contains three perturbations with *like her dog* in their right-hand side. The probability of any one of these perturbations is still 0.5 as the following example shows:

```
TK> (create-word-perturbations "I like she dog")
(("i like her dog" WORD-SUBSTITUTION 0.5))
```

Word transposition error

A word transposition error involves multiple words. This should not be too hard to detect.

```
TK> (create-word-perturbations "I saw a dog big")
(("i saw a big dog" WORD-TRANSPOSITION 0.5))
```

6.1.3 The Māori corpus

During the creation of the perturbation module, I wrote the very small English error corpus to be able to test the system. Since Te Kaitito is multi-lingual, the system should also work for the existing Māori grammar and lexicon. Victoria Weatherall is an expert on the Māori language. She used a much larger list of common student mistakes to compile a proper error corpus for this language and tested it in the perturbation system. The corpus can be found in appendix A. She found that the system works very well in correcting erroneus input.

6.2 Integrating the perturbation module within the dialogue system

So far, the perturbation module is only considered in isolation. We have seen that it is capable of giving the 'correct' corrections for isolated input, but it is much more interesting to see it actually work in the context of a dialogue. First I will talk about the way in which the system gives its feedback, then I give some results in the form of actual dialogues, including comments.

6.2.1 Providing feedback about detected errors

The focus of my project was to generate and select perturbations of input sentences. But in a CALL system, the selection of an alternative for the student's input should not be done silently. On the contrary, if the system has got enough evidence to assume the student made an error in his input, it should provide the student with feedback. As I said in section 1.3, this feedback was not the focus of my project. The simple feedback that the system gives at this point consists of the original input, the corrected input and a message telling the student to try again. In the student's original input, the flanking words (the word(s) in which the error was detected) are highlighted by means of asterices.

Naturally, this response method is just a start. All the information gathered by the perturbation process is still available and can be used to give a better teaching response. The information that is available contains the position of the flanking words, the exact perturbation type and even detailed semantic and syntactic information.

In the examples in this section, Te Kaitito is very polite to the student. To make it a little more interesting, I added a variable called *rude* which can be set to have the system give a less polite response. In section 6.2.5 on undesirable system behaviour, I set this variable to rude. For all the other examples, I used the more polite version of the feedback component.

6.2.2 Using the parser to filter syntactically incorrect perturbations

The simplest case of the perturbation system working in its environment is when the original input is grammatically incorrect and only one perturbation survives the parsing process. To show that only one perturbation survives the parsing process, I set the variable *pert-threshold* to 1 so that the perturbation module will report every perturbation that could be parsed.

I will first show all the possible perturbations of the sentence *I am hapy*:

```
TK> (create-perturbations "I am hapy")
(#S(PERTURBATION
      :ORIGINAL "I am hapy"
      :PERTURBATION "i am happy"
      :TYPE DOUBLING-DELETION
      :PENALTY 0.45200863)
 #S(PERTURBATION
      :ORIGINAL "I am hapy"
      :PERTURBATION "ii am hapy"
      :TYPE DOUBLING-DELETION
      :PENALTY 0.04108138))
```

There are two possible perturbations, but the second one is syntactically ill-formed. We expect the parser to get rid of this one, leaving only the first, correct perturbation.

The following dialogue shows that this is indeed what happens.

(6.1)
S: I am hapy
TK: You said: I am *hapy*
But I think you mean: I am happy
Please try again
S: I am happy
TK: okay

On a word level, the same thing can be proven. To show this, I use a very simple perturbation corpus with only two rules, i.e. *(I like dog → I like my dog)* and *(I like dog → I like your dog)*.

The possible perturbations of *I like dog* are:

```
TK> (create-perturbations "I like dog")
(#S(PERTURBATION
      :ORIGINAL "I like dog"
      :PERTURBATION "i like your dog"
      :TYPE WORD-DELETION
      :PENALTY 0.25)
 #S(PERTURBATION
      :ORIGINAL "I like dog"
      :PERTURBATION "i like my dog"
      :TYPE WORD-DELETION
      :PENALTY 0.25)
 #S(PERTURBATION
      :ORIGINAL "I like dog"
      :PERTURBATION "ii like dog"
      :TYPE DOUBLING-DELETION
      :PENALTY 0.04108138))
```

Now, there are two likely perturbations. The perturbation module is assumed to give both word level perturbations and discard the character level perturbation.

(6.2)
S: I like dog
TK: You said: i like ** dog
But I think you either mean:
- i like your dog
or - i like my dog
Please try again

And fortunately this is exactly what happens.

6.2.3 Semantic level error correction

The following example shows how the perturbation module helps to correct an error on the semantic level. For this example to work, I set the global variable *char-factors* to $(2, 1)$ to decrease the influence of the word length on the result.

(6.3)
S: I saw my chief
TK: Ok
S: The chef is angry
TK: You said: The *chef* is angry
But I think you mean: The chief is angry
Please try again
S: The chief is angry
TK: okay
S: A chef is angry
TK: okay

The student starts by telling Te Kaitito about his chief. This information is stored in the system's dialogue context. Te Kaitito now knows that the student has a chief and that he saw him. The student's next sentence includes a reference to *the chef*, an object which has to be in the dialogue context. Of course, there is no such object and normally Te Kaitito would try to accommodate this presuppositional information, adding the necessary chef. But it is preferable for this not to happen; in Lurcock's thesis [9], if there are two interpretations of a sentence, one of which requires accommodation and one of which does not, it will prefer the latter interpretation.

One of the perturbations of the sentence *The chef is angry* is *the chief is angry*. Both of these sentences are fed into the pipeline and will survive the parsing process. The disambiguation module now takes its turn. Since the perturbed sentence can be interpreted without accommodation, it is preferred over the original sentence. In the clarification process, the perturbation module notices that the interpretation of the original sentence has been removed which triggers the system to give the teaching response instead of adding the information to the dialogue context.

This seems a good example of the system working. To show that an angry chef is acceptable for the system, the student tells Te Kaitito that a chef is angry. The perturbation *A chief is angry* is again parsed and interpreted. But this time, the disambiguation module will not remove the original sentence because both a chef and a chief will have to be accommodated. Hence, there is no reason for Te Kaitito to assume that the student made a mistake here.

6.2.4 Disambiguating using perturbation score

In the previous sections, the existing disambiguation system was used to decide what interpretation to assign to the uttereances of the user. In this section, I give an example in which the score given to the perturbation is actually used to disambiguate.

In the first example, we try to parse the sentence *The bird flyes*. First, the global variable *pert-threshold* is set to 1 to show which perturbations survived the parsing process. The global variable *char-factors* is still $(2, 1)$ to allow for shorter words to be parsed by the system.

(6.4)
S: The bird flyes
TK: You said: the bird *flyes*
But I think you either mean:
- the bird flies
or - the bird flees
Please try again

The two options that the system has to choose between are *The bird flies* or *The bird flees*. Neither on the syntactic, semantic nor on the dialogue-act level the system has a preference for either of those. The perturbation module is called for disambiguating the sentence, but because the threshold for making a decision between different perturbations is set to 1 the perturbation module does not make this decision.

Let us first look at the actual scores for the two sentences. This can be done by simply creating all the perturbations that were sent to the parser.

```
TK> (create-perturbations "The bird flyes")
 #S(PERTURBATION
      :ORIGINAL "The bird flyes"
      :PERTURBATION "the bird flies"
      :TYPE PHONOLOGICAL-SUBSTITUTION
      :PENALTY 0.5360115)
 #S(PERTURBATION
      :ORIGINAL "The bird flyes"
      :PERTURBATION "the bird flees"
      :TYPE CHARACTER-SUBSTITUTION
      :PENALTY 0.06934482)
```

Leaving out all the perturbations that are grammatically incorrect, the two perturbations have a score of 0.54 and 0.07. The first perturbation is created by a phonological substitution and is therefore preferred.

Now we set the threshold to 0, which means that only the perturbation with the highest score will be shown. Only if there are two perturbations with the same highest score, they will both be shown to the user.

(6.5)
S: The bird flyes
TK: You said: the bird *flyes*
But I think you mean: the bird flies
Please try again

Te Kaitito now gives the highest scoring and therefore most likely correction.

6.2.5 Undesirable behaviour

The preceding examples all show Te Kaitito doing the right thing. Unfortunately, this is not always the case. The interaction of the perturbation module with other parts of the system can sometimes give unexpected and undesirable results. I will give two examples in which the system incorrectly corrects the student.

Interference with the unknown word mechanism

Te Kaitito's parsing system has a mechanism for dealing with unknown words (words that are not in the lexicon). Every time an unknown word is found, the parser will assume the word to be a proper name. The next example shows the mechanism working.

(6.6)
S: A bubblygubbly walks
TK: okay
S: What walks?
TK: bubblygubbly walks

The student tells Te Kaitito about a walking bubblygubbly. As you can imagine, the system has no information on bubblygubblies. It will still accept this sentence, assuming that there is someone or something by that name. When the student asks Te Kaitito about his information regarding things that walk, it will tell him that bubblygubbly walks. It shows that the system has stored information about something previously unknown to it.

The following dialogue shows how things can go wrong because of this.

(6.7)
S: I see a telescope
TK: okay
S: The telescop is red
TK: You suck!!!
You said: the *telescop* is red
But surely you mean: the telescope is red
...loser
S: The telescope is red
TK: okay
S: My telescop is blue
TK: okay
S: My telescope is more beautiful
TK: You suck!!!
You said: my *telescope* is more beautiful than yours
But surely you mean: my telescop is more beautiful than yours
...loser
S: My telescop is more beautiful
TK: okay

The student starts by telling the system about him seeing a telescope. Then, he makes a mistake by saying that the *telescop* is red. Te Kaitito knows about a *telescope* and corrects the student on his input. The student corrects his input and the information is accepted. So far so good, but unfortunately, the student makes the same mistake again. Only this time, he is talking about his own telescope. The system has no information about the student's telescope yet and therefore has no reason for assuming that the student is actually talking about his telescope instead of his telescop (whatever that might be). Unaware of his mistake, the student tells Te Kaitito that his telescope is more beautiful. This time, he did not make the same mistake again. However, because of the unnoticed typo before, Te Kaitito believes it has information on his telescop being blue, instead of his telescope. Hence, it tells the student he should have told it about his telescop instead of his telescope. This is an undesirable interference of the unknown word mechanism with the perturbation process.

Errors in the grammar rules

This final example might be the most interesting of all. For testing purposes, I allowed the perturbation module to prefer perturbations, even when the original sentence parsed correctly. Furthermore, I set the variable *pert-threshold* to 0.3 to only allow very likely perturbations to be preferred over the original sentence.

With these settings, it seems that the perturbation module occasionally shows some strange behaviour. For instance in the following dialogue:

(6.8)
S: A girl was very happy
TK: okay
S: The girl danced
TK: You suck!!!
You said: The girl *danced*
But surely you mean: The girl danceed
...loser
S: The girl danceed
TK: okay
S: I danced
TK: You suck!!!
You said: I *danced*
But surely you mean: I danceed
...loser

The student tells Te Kaitito about a very happy girl. All seems fine, but when the student tells Te Kaitito that the girl danced, it corrects him, telling him to say that the girl danceed instead. Let us figure out what exactly happens. One of the perturbations of the input sentence is *The girl danceed*. This perturbation has a score of 0.74. If this perturbation survives the parsing process, it will be preferred over the original sentence, because it is within the threshold of 0.3 from the score for the original sentence, which is 1. So somehow, this perturbation must have been interpreted, because it did survive the process.

The parsing system has a function that gives morphological information about any word in its lexicon. I used this function to lookup the meaning of danceed.

```
TK> (lkb::morph-analyse "DANCEED")
(("DANCE" (LKB::CP_PASSIVE_INFL_RULE "DANCEED"))
 ("DANCE" (LKB::PASSIVE_INFL_RULE "DANCEED"))
 ("DANCE" (LKB::PAST_VERB_INFL_RULE "DANCEED"))
 ("DANCE" (LKB::PSP_VERB_INFL_RULE "DANCEED")))
TK> (lkb::morph-analyse "DANCED")
(("DANCE" (LKB::CP_PASSIVE_INFL_RULE "DANCED"))
 ("DANCE" (LKB::PASSIVE_INFL_RULE "DANCED"))
 ("DANCE" (LKB::PAST_VERB_INFL_RULE "DANCED"))
 ("DANCE" (LKB::PSP_VERB_INFL_RULE "DANCED")))
```

This shows that the word *danceed* is actually generated from an incorrect application of inflection rules on the verb *dance*. To see if this is indeed the case, I looked up a similar construction for the verb *like*

```
(lkb::morph-analyse "LIKEED")
(("LIKE" (LKB::CP_PASSIVE_INFL_RULE "LIKEED"))
 ("LIKE" (LKB::PASSIVE_INFL_RULE "LIKEED"))
 ("LIKE" (LKB::PAST_VERB_INFL_RULE "LIKEED"))
 ("LIKE" (LKB::PSP_VERB_INFL_RULE "LIKEED")))
```

This suggests that Te Kaitito will know what I mean when I say that I likeed him. This final dialogue shows that this assumption is correct. My utterance is correct as well. I did like Te Kaitito, but now I do not like it anymore.

(6.9)
S: I likeed you
TK: okay
S: Who did i like?
TK: you liked me

Chapter 7

Conclusions and future work

The examples in the previous chapter show that in general the perturbation system can give a very good suggestion on how to correct erroneous input. However, for the system to be effective, the percentage of incorrect corrections should be close to zero. I did not have the time to construct a good testing environment to give a profound evaluation of the system's performance and therefore I am not able to measure its performance. What I can and have tried to do is give an explanation for those situations in which the system shows less desirable behaviour. Those situations give ground to a number of suggestions on how to further improve Te Kaitito.

7.1 Feedback

The first conclusion to make is that there is still a lot of work to be done on the system's feedback. Many times in this report, I pointed out that the purpose of a CALL system is to assist students in learning a new language. The ultimate goal is for these systems to be a substitution for a real language teacher. Although the focus of my project was not on providing feedback to the user, it must be noted that this is still a very important aspect of the system as a whole.

The perturbation module certainly lays the path for giving feedback. The process of giving correct teaching feedback starts with determining the cause of a mistake. Without knowledge of the reason the error occurred in the first place we can never credibly imitate a good teacher. The perturbation module provides information about the exact perturbation type used to resolve the error. Furthermore, it can tell which word(s) are involved in the mistake, so a future teaching module can focus on exactly those words. Finally, probabilistic information about the particular correction as well as other possible corrections is available. Combining all this information, a sophisticated feedback module could be designed which will take the system closer to its ultimate goal.

7.2 Parameter tweaking

There are a number of parameters that can be set to change the behaviour of the perturbation module. There are the two parameters for estimating the likelihood of a character perturbation. The first one consists of individual type specific penalties, each of which can be set to a value between 0 and 10. The values I used seemed to work just fine, but I did not take the time to fine-tune them. A fine-tuning of the penalties will increase the number of sensible perturbations while at the same time decrease the number of nonsense ones.

The second parameter involves the length of newly created words. I used an equation for computing the score associated with the word length which seemed reasonable to me. This does not mean that it is the optimal way of assigning a score to this factor. There might be a better equation for computing this score, based on actual findings of the influence of word length on error corrections.

On a higher level, the two character level metrics can be weighed against each other. The variable *char-factors* holds the weighing factors for both metrics. Setting this variable can cause either one of the two parameters to play a much bigger role in the final perturbation score.

This method of defining parameters which can be weighed against each other is very extendible. A third parameter for measuring the likelihood of a character perturbation can be introduced without many changes. This parameter needs to be normalized to be something between 0 and 1. Then, a third factor can be added to the weighing factors and the parameter is taken into account.

7.3 The unknown word mechanism

Section 6.2.5 shows how the unknown word mechanism can interfere with the perturbation system. This sometimes leads to funny, but undesirable situations. The mechanism will assume an unknown word to be a proper name and try to parse it as such. For perturbed sentences, the mechanism is disabled because a lot of perturbations can be easily interpreted assuming the perturbed word to be a name, i.e. every perturbation of a noun will be correct.

However, when there is good evidence for one particular pertubation, we want the system to even discard the original sentence if an unknown name is accommodated. If we would apply a penalty to the accommodation of an unknown name, a perturbation might be preferred over the original sentence. This would resolve the example given in section 6.2.5.

Bibliography

[1] Emily M. Bender, Dan Flickinger, Stephan Oepen, Annemarie Walsh, and Timothy Baldwin. Arboretum. Using a precision grammar for grammar checking in CALL. In *Proceedings of the InSTIL Symposium on NLP and Speech Technologies in Advanced Language Learning Systems*, Venice, Italy, 2004.

[2] Ann Copestake. *Implementing Typed feature Structure Grammars*. CSLI Publications, 2002.

[3] Fred Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[4] Roger Garside, Geoffrey Leech, and Geoffrey Sampson, editors. *The Computational Analysis of English: a corpus-based approach*, chapter 10. Longman, 1987.

[5] Jerry R. Hobbs, Mark Stickel, Paul Martin, and Douglas D. Edwards. Interpretation as abduction. In *26th Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference*, pages 95–103, Buffalo, New York, 1988.

[6] Daniel Jurafski and James H. Martin. *Speech and language processing*. Prentice-Hall, 2000.

[7] Slava M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recogniser. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401, 1987.

[8] Karen Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[9] Pontus Conrad Lurcock. Techniques for utterance disambiguation in a human-computer dialogue system. Master's thesis, University of Otago, 2005.

[10] W. Menzel and I. Schroeder. Constraint-based diagnosis for intelligent language tutoring systems. In *Proceedings of the IT&KNOWS Conference at the IFIP '98 Congress*, Wien/Budapest, 1998.

[11] Nanda Slabbers and Alistair Knott. A system for generating teaching initiatives in a computer-aided language learning dialogue. In *Proceedings of DIALOR 2005*, 2005.

Appendix A

Māori corpus of error rules

```
("Teenaa koe a Tio","Teenaa koe e Tio")
("Teenaa koe ko Tio"," Teenaa koe e Tio")
("Teenaa koe a Tori","Teena koe Tori")
("Teenaa koe ko Tori"," Teenaa koe Tori")
("Teenaa koe e Tori","Teenaa koe Tori")
("Teenaa koe e John","Teenaa koe John")
("Tenna koe e Marama","Teenaa koe Marama")
("Kei te pehea koe","Kei te peehea koe?")
("Kei peehea koe?","Kei te peehea koe?")
("E peehea koe?","E peehea ana koe?")
("Ko wai ia ingoa?","Ko wai too ingoa?")
("Kei hea koe?","Noo hea koe?")
("Kei too kaainga?","Kei hea too kaainga?")
("Kei te hea too kaainga?","Kei hea too kaainga?")
("Noo hea too kaainga?","Kei hea too kaainga?")
("Ko wai ngaa matua o too hoa?","Ko wai ngaa maatua o too hoa?")
("Ko wai te maatua o too hoa?","Ko wai te matua o too hoa?")
("Ko wai te maatua o too hoa?","Ko wai ngaa maatua o too hoa?")
("Ko wai maa ngaa maatua a too hoa?","Ko wai ngaa maatua o too hoa?")
("Ko wai oo maatua o too hoa?","Ko wai ngaa maatua o too hoa?")
("Ko wai oo maatua maa?","Ko wai maa oo maatua?")
("Ko wai oo matua?","Ko wai too matua?")
("Ko wai too maatua?","Ko wai oo maatua?")
("Ko wai ia maatua?","Ko wai ana maatua")
("E hia ana tau o too paapaa?","E hia ngaa tau o too paapaa?")
("E hia ngaa tau o ia?","E hia ngaa tau o too paapaa?")
("E hia ia tau?","E hia ana tau?")
("E hea oo tau?","E hia oo tau?")
("E hea oo pene?","E hia oo pene")
("Tokohia ngaa tamaiti a Moana?","Tokohia ngaa tamariki a Moana?")
("Tokohia te tamariki a Moana?","Tokohia ngaa tamariki a Moana?")
("Tokohea ngaa tamariki a Moana?","Tokohia ngaa tamariki a Moana?")
("Tokohia oo pene?","E hia oo pene?")
("Tokohia oo teina?","Tokohia oo teeina?")
("Tokohia too teeina","Tokohia oo teeina?")
("Tokohia too teina","Tokohia oo teeina?")
("Kei haere koe ki hea?","Kei te haere koe ki hea?")
("Kei te haere koe kei hea?","Kei te haere koe ki hea?")
("Kei te haere koe ki te hea?","Kei te haere koe ki hea?")
```

```
("Kei te haere Heeni ki hea?","Kei te haere a Heeni  ki hea?")
("Kei te haere John ki hea?","Kei te haere a John ki hea?")
("Kei te haere a Heeni raaua a Tio ki hea?",
       "Kei te haere a Heeni raaua ko Tio ki hea?")
("Kei te haere a Heeni raaua Tio ki hea?",
       "Kei te haere a Heeni raaua ko Tio ki hea?")
("Kei te haere ko Heeni raaua ko Tio ki hea?",
       "Kei te haere a Heeni raaua ko Tio ki hea?")
("Kei te haere a Heeni raatou ko Tio raatou ko Hone ki hea?",
       "Kei te haere a Heeni raaua ko Tio ki hea?")
("Kei te haere a Heeni maa raatou ko Tio ki hea?",
       "Kei te haere a Heeni raatou ko Tio maa ki hea?")
("Kei te haere korua ki hea?","Kei te haere koorua ki hea?")
("Kei te haere koorua ki hia?","Kei te haere koorua ki hea?")
("Kei te haere koorua ki te hea?","Kei te haere koorua ki hea?")
("Kei te haere koe me Tio ki hea?",
       "Kei te haere koorua ko Tio ki hea?")
("Kei te haere koe ko Tio ki hea?",
       "Kei te haere koorua ko Tio ki hea?")
("Kei te haere koe me ia ki hea?",
       "Kei te haere koorua ko Tio ki hea?")
("Kei te haere koorua ko a Tio ki hea?",
       "Kei te haere koorua ko Tio ki hea?")
("Kei te haere aku matua ki hea?","Kei te haere aku maatua ki hea?")
("Kei te haere koe he aha?","Kei te haere koe ki te aha?")
("Kei haere koe ki te aha?","Kei te haere koe ki te aha?")
("Kei te haere koe ki aha?","Kei te haere koe ki te aha?")
("Kei te haere au ki te kite taku whanaunga",
       "Kei te haere au ki te kite i taku whanaunga")
("Kei roto i peeke","Kei roto i taku peke")
("Kei roto peeke","Kei roto i taku peke")
("Kei runga a ngeru i te tuanui o te whare",
       "Kei runga te ngeru i te tuanui o te whare")
("Kei runga te ngeru te tuanui o te whare",
       "Kei runga te ngeru i te tuanui o te whare")
("Kei runga te ngeru i te tuanui i te whare",
       "Kei runga te ngeru i te tuanui o te whare")
("Kei runga te ngeru i te whare tuanui",
       "Kei runga te ngeru i te tuanui o te whare")
("Kei runga te ngeru i te tuanui o whare",
       "Kei runga te ngeru i te tuanui o te whare")
("Kei runga te ngeru o te tuanui o te whare",
       "Kei runga te ngeru i te tuanui o te whare")
("Kei runga te ngeru i te whare e noho ",
       "Kei runga te ngeru i te whare e noho ana")
("Kei runga te ngeru i te whare i noho ana",
       "Kei runga te ngeru i te whare e noho ana")
("Haere te horoi","Haere ki te horoi")
("Haere ki horoi","Haere ki te horoi")
("Kua haere raaua ki hootera","Kua haere raaua ki te hootera")
("Kua haere raaua ki hea hootera","Kua haere raaua ki te hootera")
("Kua haere ki te hootera raaua","Kua haere raaua ki te hootera")
("Kua hoki mai e Mere","Kua hoki mai a Mere")
("Kua hoki mai Mere","Kua hoki mai a Mere")
```

```
("Kua hoki mai i Mere","Kua hoki mai a Mere")
("Kua haere ana paapaa o Mere","Kua haere te paapaa o Mere")
("Kua tiimata ngaa tamariki i te whawhai",
       "Kua tiimata ngaa tamariki ki te whawhai")
("Kua tiimata ngaa tamariki te whawhai",
       "Kua tiimata ngaa tamariki ki te whawhai")
("Kua tiimata ngaa tamariki ki whawhai",
       "Kua tiimata ngaa tamariki ki te whawhai")
("Kua tiimata ngaa tamariki i whawhai",
       "Kua tiimata ngaa tamariki ki te whawhai")
("He aha taaima?","He aha te taaima?")
("Te taaima he aha?","He aha te taaima?")
("He aha ngaa taaima?","He aha te taaima?")
("He waru karaka","Kua Waru karaka")
("Koata paahi te ono karaka","Koata paahi i te ono karaka")
("Koata paahi e te ono karaka","Koata paahi i te ono karaka")
("Koata i te ono karaka","Koata paahi i te ono karaka")
("Koata paahi i ono karaka","Koata paahi i te ono karaka")
("Rima meneti i te ono karaka","Rima meneti paahi i te ono karaka")
("Rima paahi i te ono karaka","Rima meneti paahi i te ono karaka")
("Rima meneti paahi ono karaka","Rima meneti paahi i te ono karaka")
("Noo whitu karaka","Noo te whitu karaka")
("He aha te taaima waea a Tio?","He aha te taaima i waea mai a Tio?")
("He aha taaima i waea mai ai a Tio",
       "He aha te taaima i waea mai a Tio?")
("He aha te taaima i waea a Tio","He aha te taaima i waea mai a Tio?")
("He aha te taaima e waea mai ai a Tio",
       "He aha te taaima i waea mai a Tio?")
("He aha te taaima i waea mai ae a Tio",
       "He aha te taaima i waea mai a Tio?")
```

Appendix B

Global variables

type	*variable name*	*range*	*default value*
Character level perturbation scoring	*penalty-unknown* *penalty-close-sound* *penalty-close-keys* *penalty-swap* *char-factors*	$0..10$ $0..10$ $0..10$ $0..10$ $(0..\infty, 0..\infty)$	9 2 1 2 $(1, 1)$
Selection of perturbations using utterance disambiguation	*prefer-original* *pert-threshold*	t or nil $(0..1)$	t 0
Feedback	*rude*	t or nil	nil

Table B.1: Global variables that can be used to fine-tune the system