A SPECIFICATION LANGUAGE FOR AGGLUTINATIVE ABORIGINAL LANGUAGES FOR USE WITH FINITE-STATE SPELLING CORRECTION

RON KEATING

A Specification Language for Agglutinative Aboriginal Languages for use with Finite-State Spelling Correction

by

Ron Keating

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

Apr, 2011

St. John's Newfoundland

Abstract

There are certain North American aboriginal languages which are in danger of becoming extinct. This is partially due to the younger generations learning more major world languages in order to communicate in an increasingly global society. Furthermore, these languages tend to have only developed writing systems relatively recently, and thus do not have a rich legacy of written works to help preserve them. In order to help alleviate this problem, certain tools are being developed to facilitate communication in those languages. One such tool that is expected to help is a digital spelling correction tool. Having such a tool will make it easier to create professional digital documents in those languages, as well as help educate speakers with regard to the proper spellings of words.

To facilitate the creation of a spelling correction software tool, this thesis proposes a simplified specification language called FSCL. Linguists can use FSCL to specify the details of natural languages in a format that is easily readable by both humans and computers without having to sacrifice any relevant expressive power, thus allowing linguists and even language speakers to build and maintain a working model of the natural language in question. The syntactic and semantic details of FSCL are discussed, and an implementation built on existing finite-state natural language processing algorithms is detailed and tested with respect to a set of actual language data from one such aboriginal language, Innu.

Acknowledgements

I would like to thank my supervisor Todd Wareham for his patience and support. Without his help this thesis would not have been possible. I would also like to thank my co-supervisor Ed Brown for his frank feedback. His perspectives helped very much to focus and streamline this thesis.

Furthermore, I would like to thank my thesis examiners Carrie Dyck and Antonina Kolokolova for their constructive and thorough evaluation.

I would as well like to thank the consultants from the Linguistics department at Memorial University that helped with this project, particularly Marguerite Mackenzie and Laurel-Anne Hasler. Their patient and friendly input and feedback was critical to the results in this thesis.

In addition, I would like to thank Don Craig for his valuable help with the typesetting of this thesis in LaTeX.

Last but not least, I would like to thank my friends and family for their encouragement and moral support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Certain languages spoken in aboriginal communities are in danger of becoming extinct due to lack of speakers, such as the language spoken by the Innu [19]. One reason for this is that the number of members of those communities is quite small to begin with. Another reason is that while the number of older speakers of those languages who grew up in the traditional communities is dwindling, fewer and fewer of the younger generation who grew up in a more westernized environment are learning the native language. Furthermore, many such languages have only developed a writing system relatively recently [10]. Up until then, their traditions, stories, and culture had to be passed on verbally. Even now that they have a writing system, it will be quite some time before they have a rich written legacy. Without such a written tradition, the fewer people that know the language, the greater the danger that the language will become extinct completely. The more value the world places on literacy and text-based communication, the greater the chances that upcoming generations will abandon their traditional, primarily verbal-based languages in favor of more major world languages such as French and English.

There are certain language tools whose existence would help speakers of such endangered languages with continuing written communication in those languages. For example, orthography tools are important for representing symbols in the language. As writing in the modern world is typically done digitally, it is important for writers of the language to be able to easily represent the letters in their language. Standardized basic language aids such as dictionaries, and grammar references are also particularly useful, since these languages, as is typical of languages that are primarily verbal, experience a high degree of dialectalization, and having such standards would facilitate communication across dialects. Furthermore, spelling correction tools would help educate writers of the language with regard to the standardized spelling conventions.

Due to the small size of the speaker base of such languages, it would not be profitable for large companies with access to many resources to create spelling correction tools for these languages. As a result, any tools created for such a purpose must be designed with the assumption that there will not be teams of people who can rigorously scour code, and there will likely be only a small handful of linguists entering the language data for the spelling correction system. Thus, the system must be as easy to interact with as possible for those users.

1.2 Objectives

The primary objective of this thesis is to demonstrate that a robust specification language for natural languages can be created for aboriginal languages; in particular, agglutinative aboriginal languages like Innu. This will be demonstrated by constructing such a specification language and testing it with known finite state automaton construction and spelling correction algorithms.

The contributions made by this thesis towards this goal are:

- **A specification language** called FSCL, which is well-suited to the types of language features commonly seen in the target natural languages;
- **An interpreter** to convert languages specified in FSCL into finite-state automata;
- **A spelling correction system** which is an implementation of a common type of algorithm in the existing literature which can operate on finite-state automata; and
- **A data set of Innu nouns** which is used to test all of the above.

FSCL and its interpreter are the primary original contributions of this thesis; the spelling correction system and Innu language data are not original but were required to be implemented in order to test and evaluate FSCL.

1.3 Organization of Thesis

In Chapter 2, the background necessary for understanding the concepts and terminology employed in the remainder of the thesis will be explained. Section 2.1 gives a fundamental overview of natural language properties. Section 2.2 provides an overview of natural language processing. Section 2.3 introduces the concept of finite-state models, a paradigm on which many of the spelling correction algorithms in the literature, and particularly the one ultimately implemented in the system proposed in this thesis, are based.

In Chapter 3, the design of the specification language will be described. First, Section 3.1 gives a description of LEXC [7], an existing specification language from the literature on which many of the concepts in FSCL are based. Then, Section 3.2 describes the proposed specification language FSCL, and its similarities to and differences from LEXC.

In Chapter 4, the implementation used to test the capabilities and appropriateness of the specification language will be described. Section 4.1 describes the spelling correction algorithm used, and Section 4.2 describes the algorithm by which a specified language can be converted into a finite-state format that can be read by the described spelling correction algorithm.

In Chapter 5, the results of the test will be described as well as the conclusions that can be drawn based on those results. Section 5.1 describes the results of the testing. Section 5.2 proposes some possible directions this research could take in the future. Section 5.3 provides a conclusion.

Chapter 2

Background

This chapter will define and discuss the terms and concepts which the reader should become familiar with in order to understand the rest of the thesis. Section 2.1 will discuss natural languages, including their components, and the processes by which words are formed in natural languages. Section 2.2 will discuss natural language processing, including a discussion of different types of reference materials that aid in such processing, as well as the different types of tasks that are often performed on natural languages. Finally, Section 2.3 will contain a description of the finite-state formalism, which is a convenient method for encoding dictionaries for use with natural language processing tasks. This description will include both finite-state automata, and finite-state transducers.

2.1 Natural Languages

A natural language is a spoken and/or written language which humans use to interact with each other. English, French, Turkish, and Japanese are all examples of natural languages. This section will discuss several properties of natural languages with respect to their relevance to this thesis. The classical linguistic breakdown of components of natural languages will be introduced (Section 2.1.1), then some common word-forming processes will be discussed (Section 2.1.2) with particular emphasis on a process called affixation. For a basic background in linguistics, see [1, 27].

2.1.1 Natural Language Components

Linguists normally divide the study of natural languages into five main components:

- **Phonetics**: This is the study of the sounds involved in the speaking of a language.
- **Phonology**: This is the study of groups of sounds and how they interact with each other.
- **Morphology**: This is the study of **morphemes**, the units of meaning that can combine to form words, and how they interact with each other.
- **Syntax**: This is the study of how words can be ordered to form phrases and sentences.
- **Semantics**: This is the study of the meanings of words and sentences, given their contexts.

The processes of word construction and spelling correction fall squarely in the realm of morphology. Thus, that will be the main area with which this thesis will be concerned.

2.1.2 Word Formation Processes

Words in natural languages are formed by combining morphemes together. There are multiple processes that languages can use to do this,[1] but one of the most common and relevant to this thesis is called **affixation**. Affixation is a process by which a new word is created by combining a unit of meaning known as a **stem**, with one or more other units of meaning, called **affixes**. For example, in English, one can begin with a stem such as "walk", and add the past tense affix (in this case "-ed") to create a new word, "walked".

Natural languages that make use of affixation usually have many rules dictating what affixes can be combined with what words. For example, the "-ed" affix above cannot be combined with a noun stem such as "keyboard" to create "keyboarded". As well, an affix may take different forms depending on the form of the word it is attaching to. For example, attaching the English pluralization affix "-s" to the noun stem "church" does not result in "churchs" but rather an 'e' is prepended to the "-s" affix, resulting in "churches". Likewise, when attaching the affix "-ed" to "rate", the initial 'e' in the "-ed" affix is dropped, resulting in "rated" rather than "rateed".

Affixes can be attached to words in different positions. An affix that attaches to the beginning of a word is called a **prefix**. For example, "re-" is a common English prefix, used in words such as "remake", "review", or "redo". An affix that attaches to the end of a word is called a **suffix**. For example, "-er" is a common English suffix, used in words such as "walker", "player", or "banker".

[1] One example of such a process is **blending** [15]. This process takes part of one morpheme and part of another morpheme to create a new word. An example of blending from English would be the word "spork", which blends the first part of "spoon" with the last part of "fork".

Most languages are limited to prefixes and suffixes only. However, there are some that have affixes called **infixes** that attach into the middle of a stem. An example of this process is found in Tagalog, a language spoken in the Philippines. The infix "-um-" is inserted near the beginning of a verb stem to indicate the active voice. For example, Tagalog borrows the word "graduate" from English and to express it in the active voice (i.e. to express the idea of one *graduating* as opposed to one *being graduated*), one inserts the "-um-" infix to produce "grumaduate" [39].

Different natural languages incorporate affixation to different degrees. Some use it seldomly, others often. Some allow many affixes to attach to a single word, others few. A language that makes extensive use of affixation, both in terms of frequency of use and number of allowable affixes per stem, is called an **agglutinative language**. Turkish, Finnish, and Innu are examples of natural languages that are considered to be very agglutinative. The system being presented in this thesis is designed in a way that mainly facilitates the encoding of agglutinative languages, in particular, agglutinative North American aboriginal languages such as Innu.

2.2 Natural Language Processing

Once one is familiar with what a natural language is, the next concept that must be understood is natural language processing. It is one thing to be able to communicate in a natural language, but sometimes one wishes or needs to perform analytical tasks on that language, either manually or with the aid of an automated system. This section will first introduce the types of natural language reference materials on which such tasks operate (Section 2.2.1), followed by a more detailed description of what a natural language processing task is (Section 2.2.2), with particular emphasis on the task of spelling correction. For a basic background in natural language processing, see [14].

2.2.1 Reference Materials

There are several different types of reference materials which may exist for a particular natural language which specify different aspects of that language as comprehensively and unambiguously as possible. Some examples of such materials are as follows:

- **Lexicon**: A lexicon is a list of words or morphemes in a language [7].
- **Dictionary**: A dictionary is a list of complete words in the language, usually (though not necessarily) associated with descriptions of their meanings [29].
- **Grammar**: A grammar is a description of the ways in which words can be combined to form sentences in the language in question. It can sometimes include morphology as well as syntax [31].

These references can be paper-based, but digital versions have been becoming increasingly common.

For example, for many years the Oxford English Dictionary existed only in paper form. However it now exists in digital form as well (`http://dictionary.oed.com/entrance.dtl`). There are even tools online that search many different digital dictionaries to produce a myriad of definitions, such as `dictionary.com`.

Having digital versions of these reference materials makes it possible to create automated tools to perform processing tasks on natural languages. One of the more common reference materials that are operated on by natural language processes is the dictionary. Digital dictionaries can be encoded in a variety of ways [17, pp. 380–385]. Some common ones are:

- **N-Grams** [17, 21]: In this approach, the complete dictionary itself is not encoded, but rather, groups of n letters (where n is usually 2 or 3). Associated with each group of letters, or n-gram, is a number which indicates the likelihood of that sequence of letters occurring in that particular language. Often, a probability matrix is also encoded which indicates the likelihood of each n-gram occurring after each other n-gram.

- **Hash Table** [17, 25]: A hash table stores entries in a table such that the index of a given entry is a function of some characteristic related to that entry. An example might be a table of phone numbers, where each phone number's index in the table is some function of the name of the person who has that phone number. The function used to determine the table index is called the **hash function**. The piece of information that serves as input to the hash function is called the **hash key**. It is often difficult or impossible to design a hash function that will map each hash key to a unique table index, and thus it is possible that the function result of two different keys could be the same; this event is called a **collision**. Since two different pieces of data cannot share an index, hash tables generally have some secondary method of computing the actual indices of entries that collide; this process is called **secondary probing**.

- **Finite-state** [13, 17]: In this approach, the dictionary is encoded as a finite state automaton or finite state transducer. The finite-state approach to dictionary construction is explained in more detail in section 2.3.

Dictionary encodings can be created in a fixed manner, where the creator of the dictionary more or less encodes it directly without necessarily facilitating the modification of the list of words the dictionary represents. In situations where the language being encoded is well known and well established, and the need for words to be added, changed, or removed is expected to be rare, creating an encoding this way can be advantageous, as it can be hard-coded to take advantage of certain established features of the language to improve efficiency. However, for languages which are still being researched, for which additions and corrections to the word list the encoding represents are expected to be frequent, a more dynamic approach to creating the encoding may be more convenient.

One way to do this is to maintain a word list from which the encoding is algorithmically constructed. For some languages, a simple sequential word list may be acceptable, but for an agglutinative language, such an approach would be highly impractical due to the combinatorial nature of such languages and the resulting explosion in the number of words. For example, simply adding a root word would mean adding not only that root word alone, but also every combination of that word with every subset of allowable affixes to that word.

A more appropriate way to dynamically encode a word list for an agglutinative language is to create a special-purpose grammar for specifying words and affixes in the natural language in question, as well as the ways in which they can be combined. Such a grammar will henceforth be referred to as a **specification language**. Descriptions of natural languages in a specification language are typically much more compact than exhaustive word lists. For example, one of the more expressively powerful specification languages currently in use is called LEXC [7]. In Section 3.2 this thesis will propose a new specification language called FSCL that is particularly suited to agglutinative languages, and compare FSCL to LEXC.

2.2.2 Natural Language Processing Tasks

Any automated or computer-assisted activity that operates on a natural language is known as a natural language processing task. Some examples of such tasks include **morphological analysis** (the automatic parsing of words into their component morphemes) [16, 29], **part-of-speech tagging** (the automatic categorization of words based on their part-of-speech) [8, 28], and **spell checking** (the automatic verification of whether a given string is a correctly spelled word in the language) [17].

Another important natural language processing task is **spelling correction** [17]. Spelling correction is similar to spell checking in that a given word is checked to see whether that word is a correctly spelled word in the language or not, but rather than outputting a simple affirmative or negative response, the output instead consists of either an affirmative response or a list of correctly spelled words from the language that are similar to the input word. This is helpful to users because it provides a set of correctly-spelled words that they may have intended to type. These corrections can provide a reference in case the user did not know the correct spelling of the word. For example, if the word "phat" were input into a spelling correction algorithm for the

English language, a possible output list might be "fat", "hat" "pat", "that", "what", and "chat"

There are several approaches to spell checking and spelling correction available, which depend on the nature of the encoding of the dictionary:

- **N-Gram Analysis** [17]: To determine whether a given word is correctly spelled, every sequence of n adjacent letters in the word would be examined and that sequence would be checked against the set of n-grams. If one or more sequences are not found at all, they are tagged as errors, and suggestions with the erroneous sequences are replaced by likely alternatives from the list are output. If a particular sequence is found in the list but is very unlikely, a warning may be output along with some suggestions.

- **Hash Table** [17]: The word itself is treated as the hash key, and each table entry would be a "true" or "false" value. If the function result maps to an entry whose value is "true", the word is found in the dictionary and thus correctly spelled; if it maps to a "false" entry, the word is not found and is thus incorrectly spelled. While hash tables are normally only used for spell checking as opposed to spelling correction, it could be possible to design a hash function in such a way that similarly spelled words map to table indices that are close to each other while words that are spelled very differently from each other map to table indices that are far from each other. Given such a function, one could perform spelling correction by, upon receiving an invalid key, returning all the "true" entries within a certain proximity of that entry as suggestions.

- **Finite-state** [7, 24]: A given word is fed as input to the automaton or transducer, and considered correct if the automaton or transducer accepts that word. Otherwise, suggested corrections are output based on either the backtracks needed to accept the word, or the results of rewrite rules employed. The process of generating corrections based on backtracking is discussed in more detail in section 4.1.2.

Each of these approaches has advantages and disadvantages relative to the encoding of agglutinative languages:

- The main advantage of the n-gram method is that since only combinations of a small number of letters are being encoded, the list would be extremely light on memory in relation to other approaches which encode the entire language. However, a significant disadvantage is that, due to its probabilistic nature, this method cannot give a definitive answer as to whether the given word is indeed in the language or not; it can provide a good guess, but it may turn out to be incorrect. For example, such a technique could be fooled into a false positive by a word that contains common letter sequences but is not actually in the language. Similarly, it could be fooled into a false negative, perhaps by an odd exceptional word or a loan word historically imported from another language, that uses letter sequences not normally allowed in the language. For example, in the English word "tsunami" (a loan word from Japanese), the sequence 'ts' is not normally allowed at the beginning of an English word, and thus would likely be erroneously rejected.[2]

[2]N-Gram analysis is normally better-suited to another natural language processing task called

- The main advantage of the hash table method is its speed. Very few comparisons need to be made in order for it to arrive at a decision. However, it can be difficult to devise a hash function that minimizes collisions without greatly inflating the size of the table (let alone one that takes into account the similarity of the spellings of the words). Therefore, as an agglutinative language will have a large number of words in its dictionary, finding a suitable hash function is likely to be quite a daunting task.

- The finite-state method, while it may not be as space-efficient as n-grams or time-efficient as a hash table, does have the advantage that it lends itself to dynamic construction from a word list or specification language (Chapter 4 discusses the details on how this is accomplished). This is of particular importance for aboriginal languages, since their underlying dictionaries are more likely to require modification.

Given the above, it would seem then that the best approach to spelling correction for an agglutinative aboriginal language would be the finite-state method. Indeed, a finite-state approach is commonly used in existing spelling correction systems for other agglutinative languages such as Finnish and Turkish [24]. Hence, the approaches to spelling correction which will be discussed and compared later in this thesis are finite-state.

text recognition, whereby an image of a page of text is scanned and a text encoding of that page is output. In such a task, since scanned printed text can result in ambiguities, even for human readers (for example, it can sometimes be difficult to distinguish an "S" from a "5"), it can be helpful to compare the likelihoods of the possible letter sequences in question [17].

2.3 Finite-State Machines

Once one is familiar with the concept of natural language processing, and in particular spelling correction, the next concept that must be understood is that of the finite-state formalism on which the spelling correction algorithms described elsewhere in this thesis will be primarily based. This section will describe finite-state automata (Section 2.3.1) and finite-state transducers (Section 2.3.2) in terms of their syntax and semantics, and their differences will be highlighted. For a more comprehensive background in finite-state automata and transducers than the one presented here, see [7, 26, 29]

2.3.1 Automata

A **finite-state automaton** (FSA) is an abstract mechanism, specified relative to a formal language L, where a **formal language** is a set of strings relative to some alphabet.[3] An FSA takes a string of symbols as input and produces a binary result: "accept" if the string of symbols is a string in L, or "reject" if it is not. An FSA consists of a set of states, exactly one of which is a starting state and one or more of which are final states, and a transition function which describes how the automaton changes states during execution based on the current state and the current symbol being examined in the input string.

[3]Note that, unlike a natural language, a formal language only represents the surface forms of words without any regard to their underlying semantics.

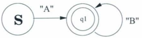
Definition 1 *A finite-state automaton is defined by the 5-tuple* $(Q, s, F, \Sigma, \delta)$*, where*

- $Q = \{q0, q1, q2, \ldots\}$ *is a set of states;*
- $s \in Q$*, is the starting state;*
- $F \subseteq Q$*, is the set of final states;*
- Σ *is the set of symbols comprising the input alphabet; and*
- δ*, a partial function of the form* $\Sigma \times Q \to Q$*, is the transition function.*

FSAs are often represented using directed graphs, where the nodes of the graph represent the states of the FSA and are labeled as such, while the edges represent transitions and are labeled with the corresponding input symbols. If the graph of an FSA contains one or more cycles, it is known as a **cyclic** FSA (see Figure 2.1). If the graph contains no cycles, it is known as an **acyclic** FSA (see Figure 2.2). The FSAs being dealt with in this thesis are acyclic.

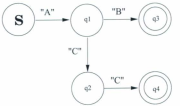
To trace the execution of an FSA on a given string, begin in the starting state, examining the first symbol of the input. Then, consult the transition function for that combination of state and symbol to determine the next state. The automaton then changes to that state and the next symbol of the input is examined to determine the next state and so on. This process continues until one of the following conditions is met:

- There are no further symbols in the input to examine. In this case, if the current state is a final state, the automaton outputs "accept"; otherwise it outputs "reject".



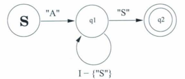
δ	s	q1
"A"	q1	–
"B"	–	q1

Figure 2.1: A cyclic FSA that accepts any word that starts with "A" and is then followed by any number of "B"s. Note that in the finite state automata diagrams in certain subsequent figures (e.g. Figure 2.3), a shorthand notation is used to combine transitions that have different symbols but go from the same source state to the same destination state – namely, a single transition will be labeled with set notation describing the set of symbols in question.



δ	s	q1	q2	q3	q4
"A"	q1	–	–	–	–
"B"	–	q3	–	–	–
"C"	–	q2	q4	–	–

Figure 2.2: An acyclic FSA that accepts an "A" followed by a "B", or an "A" followed by two "C"s.



δ	**s**	**q1**	**q2**
"A"	q1	q1	–
"B"	–	q1	–
"S"	–	q2	–

Figure 2.3: An FSA that accepts any word that starts with "A" and ends with "S", but with no intermediate "S" characters

- An input symbol is encountered for which there is no transition from the current state. In this case, the automaton outputs "reject".

For example, tracing the automaton in Figure 2.3 with the input string "areas" would result in the following sequence of steps:

1. Let us call the starting state $q0$, the intermediate state $q1$, and the final state $q2$. We will also refer to the current state as q, and the symbol in the input word which is currently being read as l. We begin with $q = q0$ and $l =$ 'a'.

2. Since $\delta(q0, \text{'a'})$ exists, we continue by setting q to $\delta(q0, \text{'a'})$ which in this case is $q1$, and setting l to the next input letter, which in this case is 'r'.

3. Since $\delta(q1, \text{'r'}) = q1$, we continue by setting q to $q1$ (no change in this case), and l to 'e'.

4. It turns out that $\delta(q1, x)$ exists and is equal to $q1$ for any letter $x \in \Sigma$ other than 's' so we may leave $q = q1$ until we read either an 's' or a symbol not in Σ. Thus, for the purposes of this example, we can read the current 'e' and the next 'a' without any state change. This leaves us reading an 's'.

5. Since $\delta(q1, \text{'s'}) = q2$, we now have $q = q2$ and the input has been completely consumed. Since $q2$ is a final state and there is no more input, the input word is accepted.

Conversely, if the same automaton were traced with the input string "ant", a different result would arise:

1. As before, we start with $q = q0$ and $l =$ 'a'.

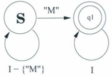
2. Since $\delta(q0, \text{'a'}) = q1$, we set $q = q1$ and $l =$ 'n'.

3. Next we set q to $\delta(q1, \text{'n'}) = q1$ and $l =$ 't'.

4. Finally, we have $\delta(q1, \text{'t'}) = q1$ so we set $q = q1$ and the input has been completely consumed. Notice that this time, the input has been consumed but we are not in a final state (since $q = q1$ which is not final). Thus, the automaton rejects the string.

It is possible to combine several FSAs together such that the final states of one FSA in the sequence are equivalent to the start states of the next. The combined FSA would then accept any string that can be expressed as the concatenation of a string accepted by the first FSA in the sequence followed by a string accepted by the second FSA in the sequence, and so on for each FSA in the sequence. This process of combining FSAs in such a way is known as **composition**. To compose two FSAs together, first create the union of the sets of states. Next, create the union of the transition functions, adding transitions from each of the final states of the first FSA to the starting state of the second FSA, whose input symbol (and output symbol in the case of FSTs) is the empty string. Finally, make the final states of the first machine non-final, and set the starting state of the composed machine to the starting state of the first machine. For example, consider the FSAs given in Figures 2.3 and 2.4. If these automata were to be composed together, the result would be the automaton in Figure 2.5.[4]

[4] The observant reader may notice that the composition operation results in an FSA that has one or more transitions labeled with the empty string. Such an FSA is **nondeterministic** [18].



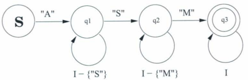
δ	s	q1
"M"	q1	q1
I - {"M"}	s	q1

Figure 2.4: An FSA that accepts any word that contains at least one "M".

FSAs are generally used to determine whether a given string is a member of a particular language. This can have many uses in natural language processing [29]. They can be employed in spell checking, whereby the FSA represents the language's

Deterministic FSAs only allow each symbol to have one transition (though a transition may be labeled with multiple symbols), and each transition must have at least one symbol in its label. In contrast, nondeterministic FSAs allow multiple transitions with the same label, as well as allowing transitions to have no label (which can be traversed without reading further input).

Conventional computers can only execute deterministic algorithms; thus, if an algorithm that is expressed using nondeterminism is to be executed by a computer, that algorithm must first be converted to a deterministic one. While composing FSAs as described here technically introduces nondeterminism with the introduction of one or more transitions labeled with the empty string, there are standard ways to convert these to deterministic automata [26]. Thus, it is safe to say that the FSAs dealt with in this thesis are all equivalent to deterministic FSAs.



δ	s	q1	q2	q3
"A"	q1	q1	q2	q3
"S"	–	q2	q2	q3
"M"	–	q1	q3	q3
I - {"A", "S", "M"}	–	q1	q2	q3

Figure 2.5: An FSA that accepts any word that starts with an "A", and contains at least one "S" which is at some point followed by at least one "M".

dictionary and if a given input word is accepted by the FSA, it is found in the dictionary and thus is correctly spelled. If it is not accepted, it is not found in the dictionary and thus is not correctly spelled. They can even be employed in spelling correction [33]. One way to employ spelling correction with FSAs is to use an algorithm that traces the FSA with the input word, but does not reject the word when faced with an input symbol for which there is no transition in the FSA. Rather, the trace would backtrack one or more steps in the FSA, taking a different transition from the one described in the input, to see if a way exists that the FSA could accept the rest of the word. The results of backtracking are then output as the suggested corrections. The spelling correction technique used in conjunction with the specification language proposed in this thesis employs an FSA in this manner.

2.3.2 Transducers

A **finite-state transducer** (FST) is a variation of an FSA that recognizes a relation R between two languages L_1 and L_2. There are several modes of operation a transducer can have:

1. Given $x \in L_1$ and $y \in L_2$, produce a binary "accept" or "reject" output, based on whether or not $(x, y) \in R$.

2. Given $x \in L_1$, produce $\{y \mid (x, y) \in R\}$

3. Given $y \in L_2$, produce $\{x \mid (x, y) \in R\}$

Only the second mode above is relevant for the purposes of this thesis.

The transition function for an FST, rather than being from a combination of state and symbol to a state, is from a combination of state and input symbol to a combination of state and output symbol.

Definition 2 *A finite-state transducer is defined by the 6-tuple* $(Q, s, F, \Sigma, \Delta, \delta)$, *where*

- $Q = \{q0, q1, q2, \ldots\}$ *is a set of states;*
- $s \in S$, *is the starting state;*
- $F \subseteq S$, *is the set of final states;*
- Σ *is the set of symbols comprising the input alphabet;*
- Δ *is the set of symbols comprising the output alphabet; and*
- δ, *a function of the form* $\Sigma \times Q \rightarrow \Delta \times Q$ *is the transition function.*

Like FSAs, FSTs are often represented using directed graphs. The nodes represent the states of the FST and are labeled as such, while the edges represent transitions and each is labeled with the corresponding input symbol or symbols, a separator symbol of some kind, and the corresponding output symbol or symbols. An example FST is given in Figure 2.6. If the graph of an FSA contains one or more cycles, it is known as a **cyclic FST**; otherwise, it is known as an **acyclic FST**.

To trace the execution of an algorithm using an FST, begin in the starting state, examining the first symbol of the input. Then, consult the transition function for that combination of state and symbol to determine the next state and the next output symbol. The automaton changes to that state and appends that symbol to the output.

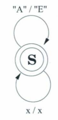


Figure 2.6: An FST that changes all "A"s in the input string to "E"s. The x/x notation above indicates that for any input symbol x (other than "A" since it has another transition), output that same symbol x.

From there, the next input symbol is examined and the next state and output symbol are determined, and so on, until the entire input string has been examined.

It is also possible to compose FSTs together. A series of composed FSTs is known as a **cascade**. To create a cascade of FSTs, simply assign them an order. An input string given to the cascade first has the first FST in the order applied to it, then the resulting output string is treated as the input string for the second FST in the sequence, and so on, treating the output string of the final FST as the output string of the cascade.

Whereas FSAs are useful for determining membership in a particular language, FSTs are useful for transforming strings. One type of string transformation sometimes used in natural language processing is known as a rewrite rule. A **rewrite rule** is a rule for transforming one string into another by replacing all substrings in the input string that match a particular pattern with another substring. For example, the rule "a $\rightarrow$ o" would transform the string "hat" to "hot", or the string "mama" to "momo". Likewise, the rule "a $\rightarrow$ ur" would transform the string "hat" into "hurt" or the string "mama" into "murmur".

The natural language tool XFST, which will be discussed extensively in chapter 3, uses FSTs as its underlying mechanism. However, the approaches to dictionary encoding and spelling correction proposed in this thesis do not make use of FSTs.

Chapter 3

Design of Specification Language

When a linguist wishes to encode a natural language for use with natural language processing tasks, the nature of the specification language used to encode the natural language is an important issue. Different specification languages have different features and abilities, and hence different strengths and weaknesses in relation to each other. The natural language and processing task in question determine which specification language features are valuable and which are not.

For example, if a linguist is attempting to encode a complete language for the purpose of morphological analysis, and that language has many rare and unusual language features and exceptions, then it would be very important that the specification language be very rich in its scope of expressiveness. If the specification language being used cannot correctly encode the language in all its detail, then no matter how readable the encoding is, or how efficiently the encoding can be compiled, if it fails to meet the linguist's primary goal of encoding the entire language, it is not useful.

On the other hand, if the linguist is encoding a language, or a portion thereof, which is fairly regular, without many quirks or exceptions, for use with some linguistic processing tasks that are computationally expensive, then it would not be very important that the specification language necessarily be very rich in terms of its ability to express a wide range of language features. Rather, the specification language would suffice as long as it can encode enough to handle the language or language portion in question. More important would be the efficiency with which computational tasks can be executed on the language encoding.

Since the purpose of this thesis is to propose a specification language able to encode northern North American aboriginal languages for the purposes of spelling correction, the features of interest to this thesis are as follows:

- **Agglutinating** structures should be easy and straightforward to encode. This linguistic feature is fundamental to many aboriginal languages and forms the backbone of how words in those languages are put together [20]. If the specification language being used to encode one of these languages cannot handle agglutination in a simple, straightforward way, it would make encoding these languages an arduous task.

- **Readability** by human users, especially ones who may not have created the original language specification, is of particular importance for specification of aboriginal languages. The writing systems of aboriginal languages such as the ones this system is concerned with have only been developed relatively recently [10], and as linguists study those languages, more is being discovered about them and more words and structures are being added to their dictionaries and grammar references [19]. Therefore, it is important for an encoding of the language for spelling correction purposes to facilitate the regular addition and modification of words and structures in the existing encoding. To facilitate this, it is important that the encoding be as human-readable as possible.

- **Spelling Correction** algorithms should be easy to apply to the structures created by the specification language. Since spelling correction is the single primary language task that is intended to be performed on the language encoding, the specification language must be designed in a way that facilitates that task, and the data structures created by it must be compatible with software that can perform that task.

This chapter will introduce FSCL, a specification language created by the author of this thesis that sufficiently fulfills these criteria. The acronym FSCL stands simply for "Finite-state Spelling Correction Language". It can refer either to the specification language itself, or to the entire software system consisting of the specification language, the interpreter, and the spelling correction algorithm (described in Chapter 4). When it is not immediately clear from context, the specification language will be referred to as the FSCL language and the entire system will be referred to as the FSCL system.

The richest, most well-described specification language that is currently in common use by linguists for finite-state natural language processing tasks is LEXC. In this chapter, LEXC will be described in detail, then examined in terms of its appropriateness with regard to the criteria above (Section 3.1). The FSCL language will likewise be described and evaluated in the same manner (Section 3.2). Discussions will show that LEXC has several problems with respect to these criteria, which FSCL resolves.

3.1 LEXC

LEXC is a specification language for natural languages created by Xerox. Its purpose is to create finite-state dictionaries and lexicons for use with their finite-state natural language processing toolkit XFST. XFST stands for "Xerox Finite State Tool", and is designed to do a wide variety of finite-state natural language processing tasks on the languages specified in LEXC. For example it can be used to create finite-state tokenizers, morphological analyzers or generators, part of speech disambiguators, or shallow syntactic parsers [7, p. ix]. LEXC and XFST are designed to do their tasks in a manner that is linguist-friendly. That is, they are designed in such a way as to be easily used, read, and understood by linguists.

In this section, the details of the LEXC language will be summarized. Following the summary will be a discussion in which LEXC and XFST's appropriateness will be evaluated with regard to the criteria in the introduction to this chapter.

3.1.1 Language Description

The syntax of LEXC is described in Backus-Naur Form notation [4] in Figure 3.1. Each entity in this description is explained individually in the bullets following.

A LEXC file (`File`) describing a language is essentially a text file, divided into lexicons. If the lexicons in the file contain any multicharacter symbols, those symbols and their representations must be defined in the beginning of the file:

```
<File>               →  <MulticharSymbols><Declarations><Lexicons>
<MulticharSymbols>   →  <MulticharSymbol><MulticharSymbols> | ""
<MulticharSymbol>    →  <Letters><Whitespace>
<Declarations>       →  <Declaration><Declarations> | ""
<Declaration>        →  <Letters> "=" <RE> ";"
<Lexicons>           →  <Lexicon><Lexicons> "|" <Lexicon>
<Lexicon>            →  "LEXICON" <Letters> "//" <Entries>
<Entries>            →  <Entries><Entry> | <Entry>
<Entry>              →  <Form> " " <ContinuationClass> ";"
<Form>               →  <FormSide> ":" <FormSide> | <FormSide>
<FormSide>           →  <FormSide><FlagDiacritic><FormSide> |
                        <FormSide> "<" <RE> ">" <FormSide> |
                        <Letters>
<ContinuationClass>  →  <Letters> | #
<RE>                 →  <Letters><RE> | <Letters> ":"
                        <Letters><RE> | "(" <RE> ")" | <RE> "*" |
                        <RE> "+" | ""
<FlagDiacritic>      →  "@" <FlagType> "." <Letters> "."
                        <Letters> "@"
<FlagType>           →  "P" | "N" | "R" | "D" | "C" | "U"
<Letters>            →  ("a" | "b" | "c" | ... | "x" | "y" |
                        "z")(<Letters> | "")
<Whitespace>         →  "//" | " "
```

Figure 3.1: Syntax Description for the LEXC specification language. This description is in Backus-Naur Form notation [4]. Each entry is of the form `<Nonterminal>` → *expression*, where *expression* can consist of a sequence of nonterminal symbols (enclosed in angle brackets <>) and/or terminal symbols (enclosed in double-quotes ""), or several such sequences separated by a vertical bar (|). The nonterminal to the left of the arrow can be replaced anywhere it appears in an expression by any of the sequences on the right. In this description, the "//" symbol is used to represent a "new line" character.

- `MulticharSymbols`: This is an optional section at the beginning of a LEXC file for defining multicharacter symbols that will be used in the word forms contained in the lexicons.

- `MulticharSymbol`: Multicharacter symbols are characters in the target language's alphabet whose ASCII representations consist of more than one symbol. For example, one may wish to represent an 'Á' as a multicharacter symbol such as 'A/' or '/A'.

In addition, LEXC allows authors to create shortcuts or abbreviations to represent frequently used regular expressions (`RE`). A regular expression is a formal symbolic way of representing a pattern or a set of strings. The LEXC syntax for such expressions is described in detail in [7, pp. 45–74]. Regular expressions are equivalent in expressive power to finite-state automata [9]. If an author wishes to make use of this feature, however, these abbreviations must be declared before the lexicons, and after any multicharacter symbol declarations, in an optional section (`Declarations`):

- `Declaration`: A declaration is a name given to a regular expression which is expected to be used frequently within the LEXC file. Rather than typing out the same regular expression over and over again, one may use a declaration to give it a meaningful name and simply use the name wherever the regular expression would be used.

Lexicons are the main content of a LEXC file. They define the constituent word forms of the language being encoded, and the rules for combining them:

- `Lexicons`: This is the only required section and contains the list of lexicons from which words in the target language will be constructed.

- `Lexicon`: A lexicon is a list of word forms which can be combined with word forms from other lexicons to form complete words. LEXC requires the first lexicon to be named ROOT, but other lexicons have no name restrictions.
- `Entries`: Each lexicon contains a list of entries associated with that lexicon.
- `Entry`: Each entry in a lexicon defines a word form and its associated continuation class.
- `Form`: Since LEXC creates finite state transducers rather than simple finite state automata, it is possible to define a word form as being two-sided: an "input" or "upper" side, and an "output" or "lower" side. When the transducer is applied "up" it will recognize "lower" forms and transform them into the "upper" forms. Likewise, when the transducer is applied "down", it will recognize "upper" forms and transform them into "lower" forms.
- `FormSide`: Both the "upper" and "lower" forms define how a word or part of a word appears as its surface form and as well can include some linguistic information defining how it interacts with word forms from other lexicons under certain conditions.
- `ContinuationClass`: A continuation class represents the next lexicon that word forms can attach from, if this form is included in a word. In LEXC there is no restriction on which lexicon an entry can have as its continuation class.

The most powerful and versatile tool for providing linguistic information in LEXC is the flag diacritic (`FlagDiacritic`); particularly for handling long-distance dependencies. A flag diacritic contains information about how a word form can combine with other forms from other lexicons. As words are being created or recognized, the values of the flags are "remembered" and can be "changed" when other word forms with matching flag names are encountered.

The first part of a flag diacritic is the flag type, the second is the flag name, and the third is the flag value. There are no restrictions on what strings the user may use as a flag name or flag value:

- `FlagType`: The flag type symbol defines exactly how the diacritic behaves:
  - **P**: Set the specified flag to the specified value (overwriting whatever value it had before).
  - **N**: Set the specified flag to the complement of the specified value (overwriting whatever value it had before).
  - **R**: Reject the word if the specified flag is not set to the specified value (or if it has not been set to any value).
  - **D**: Reject the word if the specified flag is set to the specified value. The value parameter can be omitted in this type of flag diacritic, in which case, reject the word if the specified flag has been set to any value.
  - **C**: The value parameter is omitted in this type of flag diacritic. Clear the value of the specified flag.
  - **U**: If the value of the specified flag has not been set, set it to the specified value. Otherwise, if the value has been set to any value other than the specified one, reject the word.

Although LEXC's flag diacritic system is a source of great expressive power, it can also lead to problems; particularly with respect to readability (see Section 3.2.2).

3.1.2 Discussion

An analysis of the LEXC language in terms of the criteria defined in the introduction to this chapter yields the following:

- **Agglutination**: The LEXC language is quite rich. It is able to account for many language features, even ones that are very rare. A combination of LEXC's rich flag diacritic system and XFST's ability to apply finite-state filters allows Xerox's system to handle complicated processes such as infixation, reduplication, and interdigitation.

  Unsurprisingly then, agglutination is not a problem for LEXC. In fact, its system of lexicons and continuation classes puts agglutination at the very core of its operation.

- **Readability**: The LEXC language can be quite readable as long as there are not many regular expressions, flag diacritics, and operations distinguishing "upper" forms from "lower" forms, as illustrated in Figure 3.2. If these features are used extensively enough, however, the language specification can become difficult to follow, as illustrated in Figure 3.3.

  Even without regular expressions and definitions of upper and lower forms, the ability for certain types of flag diacritics to change or erase information from others in a word means that the location of a flag within a word matters

```
LEXICON Nouns
kirit     CaseEnd ;
wadil     CaseEnd ;
ridok     CaseEnd ;
faarum    CaseEnd ;
wup       NomAcc ;

LEXICON CaseEnd
u         # ;
a         # ;
i         # ;

LEXICON NomAcc
u         # ;
a         # ;
```

Figure 3.2: Example of easily readable LEXC language description [7, p. 244]

and even this can cause some clutter which hinders the readability of a LEXC description. For details, see Section 3.2.2.

- **Spelling Correction**: Surprisingly, despite the wide variety of language tasks that XFST is well-suited to, spelling correction is not one of them. The authors of the XFST documentation offer a solution involving the author of the language specification being familiar with common spelling errors in the language and creating a cascade of replacement rules [7, pp. 451–453]. However, it is highly unreasonable to be expected to create a list of every possible spelling error in a language and its corresponding replacement rule. This alone renders rigorous spelling correction using that technique out of the question.

  In addition, as the authors point out, the order in which the replacement rules are applied matters, as it is possible for one rule to undo the work done by

```
LEXICON Stems
pakai       meN- ;
paksa       meN[redup[-]] ;
payah       meN[redup[-]]kan ;

LEXICON meN-
< "+meN-":0              "@R.PREF.meN" > #;

LEXICON meN[redup[-]]
< [ "+meN[redup[-]]" .x.
    "}" "%^REHYPH" "]" "^" 2 "^]"
                  "@R.PREF.meN[redup[-]]@" > #;
  ]

LEXICON meN[redup[-]]kan
< [ "+meN[redup[-]]kan" .x.
    "}" "%^REHYPH" "]" "^" 2 "^]" k a n
                  "@R.PREF.meN[redup[-]]@" > #;
  ]
```

Figure 3.3: Example of less readable LEXC language description (abbreviated from [7, pp. 390–391])

another [7, pp. 142–143]. For example, suppose there were a rule `ph` $\rightarrow$ `f` (say, to correct "phat" to "fat"). Suppose, for the same language, there were another rule `f` $\rightarrow$ `ph` (say, to correct "fone" to "phone"). If these two rules are in the same cascade, then no matter which order they are applied in, one of the example words will go uncorrected. If `ph` $\rightarrow$ `f` is applied first and `f` $\rightarrow$ `ph` afterwards, then, given the input "phat", it will first correctly transform it into "fat", but then erroneously transform it back into "phat". Likewise, if `f` $\rightarrow$ `ph` is applied first and `ph` $\rightarrow$ `f` afterwards, then, given the input "fone", it will first correctly transform it into "phone", but then erroneously transform it back into "fone".

XFST offers a solution to this situation in the form of "parallel rules", where several rules can be applied simultaneously. However, just as one cannot simply apply all spelling correction rules in an arbitrarily ordered cascade, one cannot simply apply them all in parallel either. For example, consider a rule such as `z` $\rightarrow$ `s` intended to correct misspellings such as "hatz" to "hats". Consider also a rule such as `s` $\rightarrow$ `es` intended to correct misspellings such as "churchs" to "churches". If these rules were to be applied in parallel, then when given the input "churchz", the `z` $\rightarrow$ `s` rule would be applied, but not the `s` $\rightarrow$ `es` rule, resulting in the word being erroneously transformed into "churchs". While it would be possible, for the sake of this simple example, to merely create a rule that encapsulates both operations, such as `z` $\rightarrow$ `es`, to attempt to define such a rule for every such possible dependency in a large set of rewrite rules would be a daunting task. Therefore, this approach to spelling correction is highly impractical, not only because one must be able to produce a large set of possible corrections (and precisely define the contexts in which those corrections are to be applied), one must also pay attention to the ordering of those corrections.

At this point one may suggest that perhaps a language could still be described in LEXC and compiled into an FSA using XFST, then extract that FSA and perform other, more appropriate spelling correction techniques on it. However, it turns out that although XFST does have a built-in function that allows the user to print a finite-state machine from XFST to the screen or to a text file, and this function works perfectly on small, simple automata, it refuses to operate at all on anything large and complex enough to be interesting, let alone on an automaton large and complex enough to properly encode a natural language.

Therefore, if one were to use the LEXC language as a specification language for use with spelling correction, one would be unable to use the XFST software it was designed for and hence have to write their own LEXC compiler or interpreter. Furthermore, if spelling correction is the only task the linguist is concerned with, many of the features of LEXC that clutter its descriptions and complicate its compilation could be stripped away.

To summarize, although LEXC handles agglutination well, it falls down with respect to readability and compatibility with spelling correction.

3.2 FSCL

Whereas LEXC was designed for performing many language processing tasks on many kinds of languages, the only language task FSCL is designed for is spelling correction, and the only kind of languages it is designed for is those with an agglutinating structure. Therefore there are many features of LEXC which are not needed and therefore not included in FSCL. Furthermore, some existing features from LEXC have simplified equivalents in FSCL for the same reason. Nevertheless, morphological analysis and other language processing tasks could still be done on the finite-state automata created by FSCL language descriptions if the appropriate tools were created.

In the same manner as the previous section, this section will begin with a description of the FSCL language, followed by an evaluation with respect to the criteria in the introduction to this chapter.

3.2.1 Language Description

This section will describe the syntax structure of FSCL and explain in detail the various entities in a FSCL language description and what they mean.

The syntax of FSCL is described in BNF notation in Figure 3.4. Each entity is then explained individually in the bullets following.

A FSCL language description consists of three separate text files: one containing prefixes (`PrefixFile`), one containing word stems (`StemFile`), and one containing suffixes (`SuffixFile`):

- `Lexicon`: A lexicon represents a group of stems or affixes and the information that limits the other stems and affixes to which they can be attached.

```
<PrefixFile>             →  <Lexicon><PrefixFile> | <Lexicon>
<SuffixFile>             →  <Lexicon><SuffixFile> | <Lexicon>
<StemFile>               →  <Lexicon>
<Lexicon>                →  <Label> "//" <EntryList>
<EntryList>              →  <Entry><EntryList> | <Entry>
<Entry>                  →  <Form> ";" <ContinuationClass> "//"
<Continuation Class>     →  <Label> | #
<Form>                   →  <Letters><Flag Diacritic><Form> |
                            <Letters><Form> | <Letters>
<Flag Diacritic>         →  "@" <Alphanumerics> "." <Alphanumerics>
                            "@"
<Label>                  →  <Alphanumeric><Label> | <Alphanumeric>
<Letters>                →  <Letter><Letters> | <Letter>
<Alphanumerics>          →  <Alphanumeric><Alphanumerics> |
                            <Alphanumeric>
<Letter>                 →  "a" | "b" | "c" | ... | "z" | ""
<Alphanumeric>           →  "a" | "b" | "c" | ... | "z" | "0" | "1" |
                            "2" | ...| "9"
```

Figure 3.4: Description of FSCL syntax. This description uses the same notation conventions as Figure 3.1

Unlike LEXC, FSCL has no leading information required before the lexicons, such as multicharacter definitions or declarations. The structure of its lexicons, however, is largely the same.

- `EntryList`: An entry list is the part of the lexicon which represents the list of forms it contains and information about how each form can combine with forms from other lexicons.
- `Entry`: An entry represents a particular form and the information about how it can combine with forms from other lexicons.
- `ContinuationClass`: A continuation class represents the next lexicon that forms can attach from if this form is included in a word. Unlike LEXC, which allows any lexicon to serve as a continuation class, FSCL only allows lexicons subsequent to a particular lexicon to serve as continuation classes for the entries in that lexicon. For the entries in lexicons in the prefix file, the continuation class can be any lexicon specified later in the prefix file, or can be the stem lexicon. For the entries in lexicons in the stem file or suffix file, the continuation class can be any lexicon stated later in the suffix file, or can be the '#' reserved symbol, representing the end of the word, i.e. no further affixation permitted. The reason FSCL restricts the lexicons that can be used as continuation classes while LEXC does not, is that the spelling correction algorithms which operate on automata produced by FSCL descriptions are ones which require acyclic automata, while the ones which operate on the automata produced by LEXC descriptions evidently do not. Furthermore, FSCL is designed for languages such as Innu which have fixed ordering of affixes, while LEXC is much more

general-purpose and thus cannot afford to take this property for granted.

- `Form`: A form is the string of letters representing the surface form of a particular stem or affix in a lexicon and certain information about what other forms can attach to words containing it.

FSCL, like LEXC, also has a system for flag diacritics, albeit a much simpler one:

- `FlagDiacritic`: A flag diacritic contains information about what other forms can attach to a word containing the form it is part of. Two forms have conflicting flag diacritics if they each have a flag diacritic with the same feature but different values. If a word contains conflicting flag diacritics, it is rejected.

- `Label`: A label represents the name of a particular lexicon to identify it and enable the use of continuation classes.

Only one type of flag is supported (equivalent to the **U** type in LEXC), which leads to a slightly less robust yet far simpler system to define and to work with.

3.2.2 Discussion

An analysis of the FSCL language in terms of the previously defined criteria yields the following:

- **Agglutination**: Since FSCL uses the same type of lexicon and continuation class structure as LEXC, it too has agglutination at the very core of its operation. While there are language features that LEXC can encode and FSCL cannot, these features are rare and are not of primary concern to the goals of this system.[1] The sacrifice that including the ability to handle these features would require in terms of readability would be too great in relation to the likelihood of those features actually being encountered in the intended scope of languages this system is designed for.

- **Readability**: Much of the unreadability of complicated LEXC specifications comes from the requirement of positioning the symbols and notation representing the information about the word forms in the middle of the word forms. This can easily lead to a word form being broken up by symbols and other information and hinder the user from reading the word form itself.

  For example, a "set" flag diacritic which comes after a "clear" flag diacritic results in a very different set of words when compiled than if the "clear" flag were placed after the "set" flag. To illustrate this phenomenon, consider Figure 3.5. This LEXC description produces no strings. The first entry starts a word consisting of just "a" and sets the A flag to "true". Then, the continuation class is followed to lexicon `Second` where it combines the "a" with "b" to produce "ab" and clears the value of the A flag. Next, the continuation class is followed to lexicon `Third`, where it concatenates a "c" to the "ab" but then rejects the word on the basis that the A flag does not have the value "true" (since it has no value thanks to being cleared by the entry in lexicon `Second`). In contrast, consider the lexicons in Figure 3.6. This LEXC description produces the string "abc". The first entry starts a word consisting of just "a" and clears the A flag.

[1] For the reader interested in examples of the details of such features, the authors of LEXC have a chapter devoted to the concept of non-concatenative morphotactics which can be difficult to encode in this type of specification language [7].

```
LEXICON First
a@S.A.true@   Second;

LEXICON Second
b@C.A@        Third;

LEXICON Third
c@R.A.true@   #;
```

Figure 3.5: Illustration of the order of flag diacritics affecting the language output in LEXC.

Then, the continuation class is followed to lexicon `Second`, where it combines the "a" with "b" to produce "ab" and sets the A flag to "true". Next, the continuation class is followed to the lexicon `Third` where it concatenates a "c" to the "ab" but this time does not reject the word, since its requirement that the A flag be set to "true" has been met. Thus, since the sets of strings produced by this description and the previous one are different, and the only differences in the descriptions are the order of flag setting/clearing, it is clear that changing the order of "set" and "clear" flags can change the sets of strings produced by a language description in LEXC.

Likewise, differences between "upper" forms and "lower" forms within words must be expressed where they appear, which can cause complexity in the word form, especially when the word forms in question are long or vary in length. Consider the lexicons in Figure 3.7. Note that as long as the strings in the upper forms are the same length, the distinction of upper and lower forms is actually quite readable and organized, even if the length of the strings in the

```
LEXICON First
a@C.A@        Second;

LEXICON Second
b@S.A.true@   Third;

LEXICON Third
c@R.A.true@   #;
```

Figure 3.6: Illustration of the order of flag diacritics affecting the language output in LEXC.

lower forms differ, as illustrated in lexicon `Neat`. However, since LEXC does not allow whitespace between an upper and lower form description, lexicons whose upper-side strings vary in length can cause clutter in the description, as illustrated in lexicon `Messy`. Contrast this with the readability of the same words in lexicon **`Illegal`**, which is not possible in LEXC but would be were it to allow whitespace in its description of upper/lower form distinction.

Although LEXC's flag diacritic notation is very rich, there is only one of LEXC's many types of flag diacritics which gets used extensively, almost to the point of exclusivity: the "unification" type. Restricting the flag diacritic system to only allow unification-type flag diacritics has several benefits. Not only does it make implementation of a compiler for the language easier by having one less piece of information to keep track of, but it also makes the syntax simpler for the user to read and understand. This way, neither the compiler nor the user has to worry about the possibility of flag diacritic values being changed or cleared during word construction, thus making lexicon construction much more

```
LEXICON Neat
eat:ate         #;
win:won         #;
see:saw         #;
fly:flew        #;
sit:sat         #;

LEXICON Messy
do:did                            #;
recite:recited                    #;
jump:jumped                       #;
inspire:inspired                  #;
underestimate:underestimated      #;

LEXICON Illegal
do               :do                 #;
recite           :recite             #;
jump             :jumped             #;
inspire          :inspired           #;
underestimate    :underestimated     #;
```

Figure 3.7: Illustration of the readability of LEXC's method of distinguishing upper forms from lower forms.

straightforward for the user. Although there may be certain types of linguistic phenomena that cannot be modeled with only unification-type flag diacritics, these phenomena are rare, and the benefits of a unification-only flag diacritic system seem to far outweigh this drawback.[2]

Similarly, FSCL has no such concept of "upper" forms and "lower" forms. These were necessary in LEXC since it deals with transducers rather than recognizers. In simple FSMs like the ones created from FSCL descriptions, a word is either in the dictionary or it is not; a word cannot be transformed to another string.

Therefore, a natural language specification using FSCL can be written in a format which has complete word forms followed by all the other linguistic information. Keeping the word form separate from the other information rather than having them intermingled with each other naturally leads to a much clearer, more organized, and hence more readable notation. Figure 3.8 illustrates this difference in readability. The lexicons labeled "Messy" and "Alternative" are legal in both LEXC and FSCL, while the one labeled "Neat" is only legal in FSCL. In "Messy", the words are lined up with each other, but all the meta information is jagged. In "Alternative", some of the flags are lined up with each other, but others are not, nor are the words themselves. Additionally, depending on the nature of the words and flags being used, LEXC may require certain flags to be in certain positions relative to the word, so the user may

[2]An example of such a phenomenon is given in [7] where it can happen in certain languages with vowel harmony, such as Mongolian, that a word that would normally require front-harmony can be compounded into a front-harmony word. To accomplish this, the harmony feature must be changed during word construction.

not have the freedom to choose between these alternatives. In "Neat", on the other hand, all information of the same type can be lined up with each other vertically, making it much easier for human readers to parse at a quick glance.

```
LEXICON Messy
 sandwich@U.Animate.false@@U.PluralizeWithES.true@
 church@U.Animate.false@@U.PluralizeWithES.true@
 finch@U.Animate.true@@U.PluralizeWithES.true@
 dog@U.Animate.true@@U.PluralizeWithES.false@
 cat@U.Animate.true@@U.PluralizeWithES.false@
 cow@U.Animate.true@@U.PluralizeWithES.false@@U.Gender.Female@
 house@U.Aniamte.false@@U.PluralizeWithES.false@

LEXICON Alternative
 @U.Animate.false@sandwich@U.PluralizeWithES.true@
 @U.Animate.false@church@U.PluralizeWithES.true@
 @U.Animate.true@finch@U.PluralizeWithES.true@
 @U.Animate.true@dog@U.PluralizeWithES.false@
 @U.Animate.true@cat@U.PluralizeWithES.false@
 @U.Animate.true@cow@U.PluralizeWithES.false@@U.Gender.Female@
 @U.Animate.false@house@U.PluralizeWithES.false@

LEXICON Neat
 sandwich @Animate.false@ @PluralizeWithES.true@
 church   @Animate.false@ @PluralizeWithES.true@
 finch    @Animate.true@  @PluralizeWithES.true@
 dog      @Animate.true@  @PluralizeWithES.false@
 cat      @Animate.true@  @PluralizeWithES.false@
 cow      @Animate.true@  @PluralizeWithES.false@ @Gender.Female@
 house    @Aniamte.false@ @PluralizeWithES.false@
```

Figure 3.8: Illustration of the difference in readability achieved by the ability to separate word forms from meta information. The lexicons labeled "Messy" and "Alternative" are legal in both LEXC and in FSCL, while the lexicon labeled "Neat" is legal in FSCL but not in LEXC.

- **Spelling Correction**: There are not many algorithms in the literature that can perform spelling correction on possibly cyclic finite-state transducers such as those produced by LEXC descriptions. However, there is a wide array of spelling correction algorithms available which operate on acyclic finite-state automata (see Chapter 4).

  Therefore, since FSCL does not need to deal with the specification language features involved in creating possibly cyclic transducers, such as regular expressions (which can cause cycles) and "upper" and "lower" word forms (which require transducers as opposed to simple automata), these features can be omitted, reducing the complexity of the specification language while at the same time making it more streamlined for the production of acyclic finite-state automata, hence making it more suitable for use in spelling correction algorithms.

As has been shown above, FSCL remedies many of the problems LEXC had with respect to the criteria defined in the introduction to this chapter. Now that a specification language has been designed, the next step is to implement software that interacts with it and allows the languages defined in it to be compiled into finite state automata and spelling correction to be performed on those automata. Such an implementation will be described in the next chapter.

Chapter 4

Implementation of Specification Language

In the previous chapter, the specification language FSCL was described. However, simply having a specification language is not sufficient for spelling correction. Rather, an encoding of a natural language using this specification language will be the input for a spelling correction algorithm that must be implemented. Furthermore, the language encoding may have to be converted into a form that the spelling correction algorithm can accept as input, for which an interpreter will be required.

Hence, in order to implement spelling correction on a language encoded in FSCL, one needs the following:

- **Spelling correction algorithm**: A spelling correction algorithm appropriate for agglutinative aboriginal languages must be selected and implemented. For reasons discussed in Chapter 2, this algorithm should be one that operates on finite state automata.

- **Interpreter**: Since the spelling correction algorithms in question operate on FSAs, the FSCL description must be converted into such an automaton. For this, an interpreter is needed.

Such an implementation will help demonstrate whether spelling correction can indeed be implemented on languages encoded in FSCL in a manner that is efficient in terms of both space (computer memory) and running time.

In this chapter, Section 4.1 will discuss the implementation of the spelling correction algorithm, and Section 4.2 will discuss the implementation of the interpreter. Each section will begin with a discussion of previous work, followed by a description of the implemented algorithm (including pseudocode), and finally a discussion of the process of implementing that component.

4.1 Spelling Correction

In order for a spelling correction algorithm to be appropriate for use with FSAs created from FSCL descriptions, there are certain criteria that this algorithm should meet:

- **Comprehensive**: The algorithm should be able to correct as many misspellings as possible. If it must leave any words uncorrected, this should happen as infrequently as possible.
- **Efficient**: The algorithm should use the least time and space resources possible as a function of input size. Note, however, that in the domain of spelling correction, there are many ways in which one can measure the size of the input.

One can consider, for example, the number of misspellings within a word, the length of the word itself, or even the edit distance of the word from a word in the language. Depending on the nature of the language this operation is to be carried out on, certain variables may be of more consequence than others. For instance, in a language with very short words but whose speakers tend to produce very high numbers of misspellings within a word, one can afford to use an algorithm that sacrifices efficiency with respect to word length to gain efficiency with respect to misspelling rate.

The languages on which FSCL is designed to operate tend to have long words, and although the speakers of these languages tend to frequently misspell words, the actual number of misspellings within a word tends to be low, despite the high word lengths involved [19]. Therefore, an appropriate spelling correction algorithm to implement for languages encoded in FSCL should be as efficient as possible with respect to word length, even if it means sacrificing some efficiency with respect to the number of misspellings per word.

In the next subsection, the most common types of spelling correction algorithms in the literature are described and examined in relation to these criteria.

4.1.1 Previous Work

Most of the finite state spelling correction techniques described in the literature fall into one of the following categories:

- **Rewrite Rule**: [3, 7] This is the technique suggested by the authors of XFST for use with their software. Such an algorithm takes a language description in the form of an FST and a list of rules representing common spelling errors in that language. The algorithm takes each word to be corrected and applies the rewrite rules, then checks to see if the modified word is accepted by the FST.

- **Dynamic Programming**: [23, 36] This technique uses dynamic programming to compute the minimum error distance from the word to be corrected to a word accepted by the automaton and produce that accepted word. Here, the **error distance** between two strings w_1 and w_2 is the number of edit operations (such as deletions, insertions, or transpositions of characters) required to transform w_1 into w_2.

- **Depth-First Search**: [22, 24, 32, 34, 35] This technique involves performing depth-first search[1] on the FSA that recognizes the language with respect to the word to be corrected, allowing for a specified maximum number of correction operations. This represents a form of look-ahead and/or backtracking process in the search.

Each of these types of techniques will now be discussed in relation to the goals defined in the previous subsection.

The rewrite rule technique was not appropriate for the goals of this project. As discussed in Section 3.1.2, this method's allowance for incorrect words to remain uncorrected, not to mention the impracticality of being able to specify every type of

[1]Depth-first search [12, Section 23.3] is a standard technique for exploring the nodes of a tree which operates by recursively exploring as far down one branch as possible before returning and exploring the next branch. This is in contrast to the breadth-first search [12, Section 23.2] technique which explores all adjacent nodes first before then exploring the nodes adjacent to each of those.

spelling error one could make in the language, outweighed any benefit of efficiency.

The dynamic programming technique, while quite comprehensive, was also deemed inappropriate due to its time and space requirements. The algorithm requires both space and time proportional to $w \times s^2$, where w is the length of the word to be corrected, and s is the number of states in the automaton [36]. While the languages intended to be encoded by FSCL can have word lengths that are quite long, even these high word lengths are inconsequential in relation to the number of states. The number of states in a comprehensive automaton that accepts an entire language is quite large, even when minimized. For example, the FSA resulting from encoding only the Innu nouns had 28,163 states. This extreme resource requirement was therefore enough to reject this technique as an option.

The depth-first search technique, on the other hand, requires space proportional to $w \times f$, where w is the length of the word to be corrected, and f is the maximum fanout of any state in the automaton.[2] Since the transitions in the automaton are based on the next letter in the string, the maximum fanout any state could have in the automaton cannot be any greater than the size of the alphabet for the language. This means the number is not only far smaller than the number of states in the automaton, but is actually something practical and reasonable; indeed, it can even be smaller than the length of the word to be corrected on occasion.

The time requirement for the depth-first search technique is proportional to $\frac{w!}{d!(w-d)!} \times (w+d) \times 2^d \times f^d$, where w is the length of the word to be corrected, f is the maximum fanout of any state in the automaton, and d is the maximum error

[2] This is the maximum space needed to perform depth-first search with backtracking for a specific string on an automaton.

distance [30]. While this may seem like an unacceptably large number due to the d seen in the exponent for some of the terms, recall that the speakers of the languages with which FSCL is concerned tend to produce low numbers of errors per word. In fact, the error distance in practice is usually a very small number such as 2 or 3, thus allowing this time requirement to actually be very reasonable.

In terms of comprehensiveness, while it is true that the existence of a maximum error distance can cause a severely misspelled word to go uncorrected, it is very seldom that this occurs, even for low maximum error distance values. In addition, assuming that the automaton accepts any words, there always exists an error distance at which the given word will be corrected into a word accepted by the automaton, meaning an interface can be constructed that allows the user, upon finding a misspelled word with no suggestions for corrections, to re-run the algorithm on that word with a higher maximum error distance until a suggestion is provided.

Therefore, since both the efficiency and comprehensiveness of the depth-first search technique seem reasonable for the purposes of this project, it was chosen as the technique for the spelling correction algorithm with which to test the automata produced by FSCL language descriptions.

4.1.2 Description of Algorithm

The pseudocode for the depth-first search spelling correction algorithm is given in Figures 4.1 and 4.2. This algorithm begins by trying to see if the given word is accepted as-is, without requiring any edits. Each time it encounters a transition it cannot make within the given FSA, it recursively tries to "undo" several different types of edit operations in the word; specifically: insertion, transposition, deletion, and substitution. As it does so, it keeps track of the list of resulting corrections. The maximum error distance threshold with which the `TolerantLookup` function is called limits the recursion depth and hence the number of edits that will be permitted in a given correction. For a more comprehensive discussion and explanation of the details of this algorithm see [30].

```
Type ResultSet
   Integer error_distance
   Set suffixes

Type State
   Boolean final

Function AddOrReplace(ResultSet old, ResultSet new)
   ResultSet result
   if new.error_distance < old.error_distance then
      result <- new
   else
      result.suffixes <- old.suffixes U new.suffixes
      result.error_distance <- old.error_distance
   return result
```

Figure 4.1: Pseudocode of spelling correction algorithm. Pseudocode in this figure and the ones following use the object-oriented notation `x.y` to represent variable `y` belonging to object instance `x`. The symbol `U` is used to represent the set union operator, and the symbol `+` is used to represent either mathematical addition or string concatenation, depending on the context in which it is used. The symbol {} represents an empty set and the symbol `[]` represents an empty array.

```
Function TolerantLookup(String w, Integer wp, State s,
                        Integer t)
   ResultSet result
   Set suffix
   result.error_distance <- infinity
   if wp > w.length then
      if s.final then
         result.error_distance <- 0
         result.suffixes <- {}
   else
      if transition (s, w[wp]) exists then
         result <- TolerantLookup(w, wp + 1, (s, w[wp]), t)
         if result.error_distance < t then
            t <- result.error_distance
         for each string z in result.suffixes
            z <- w[wp] + z
   if t > 0 then
      // Insertion //
      if wp <= w.length then
         n <- TolerantLookup(w, wp + 1, s, t - 1)
         increment n.error_distance
         result <- AddOrReplace(result, n)
         if result.error_distance < t then
            t <- result.error_distance
      // Transposition //
      if wp <= w.length and w[wp] != w[wp + 1] then
         if transition (s, w[wp + 1]) exists then
            if transition ((s, w[wp + 1]), w[wp]) exists then
               n <- TolerantLookup(w, wp + 2,
                              ((s, w[wp + 1]), w[wp]), t-1)
               for each string z in n.suffixes
                  z <- w[wp + 1] + w[wp] + z
               increment n.error_distance
               result <- AddOrReplace(result, n)
               if result.error_distance < t then
                  t <- result.error_distance
```

Figure 4.2: Pseudocode of spelling correction algorithm (cont'd)

```
    for each letter l in the alphabet
        if transition (s, l) exists then
            // Deletion //
            n <- TolerantLookup(w, wp, (s, l), t - 1)
            for each string z in n.suffixes
                z <- l + z
            increment n.error_distance
            result <- AddOrReplace(result, n)
            if result.error_distance < t then
                t <- result.error_distance
            // Replacement //
            if wp <= w.length then
                n <- TolerantLookup(w, wp + 1, (s, l), t - 1)
                for each string z in n.suffixes
                    z <- l + z
                increment n.error_distance
                result <- AddOrReplace(result, n)
                if result.error_distance < t then
                    t <- result.error_distance
return result
```

Figure 4.3: Pseudocode of spelling correction algorithm (cont'd)

4.1.3 Discussion

Although there were no problems with the implementation of this module of the system, there are some ways in which it could have gone wrong if certain definitions had been taken for granted rather than verifying that they match up with those in the field of linguistics:

For example, most Computer Science papers on this topic define "misspellings" in terms of Hamming distance, or some slight variation thereof, where the Hamming distance between two strings is the number of positions at which those strings are different [35]. While this definition turned out to be acceptable for the linguist consultants on this project, it would have been dangerous to take for granted that this was the case, because there are many ways to define misspellings, and many factors which may influence the weighting of errors. For example, the position of the misspelling in a word could have affected how significant an error it was. Likewise, changing a consonant into a vowel or a vowel into a consonant could have been more significant or less significant than changing a consonant into another consonant or a vowel into another vowel. It is also possible that deleting or changing an entire syllable could have had less severity than deleting or changing part of it.

This simply serves to emphasize the fact that in Computer Science, we should never assume that the definitions we use for certain concepts are shared by those in other fields.

4.2 Interpreter

The spelling correction algorithm discussed in Section 4.1 operates on finite-state automata. However, FSCL descriptions in their raw form are stored as flat text files. Thus, an interpreter must be implemented to convert a FSCL language description into an FSA that can be used by the spelling correction algorithm.

A literature search did not reveal any existing algorithms that take compact language descriptions such as those in FSCL encodings and turn them into FSAs directly; however there were many which turn flat word lists into FSAs [13, 37]. Therefore the simplest solution was to implement such an algorithm, but first pre-process the FSCL description into a word list.

Such an algorithm converting a word list into an FSA must meet the following criteria in order to be considered appropriate for FSCL:

- **Space efficient**: Since one does not know ahead of time how large the word list will be, it is important that the algorithm be space efficient to ensure that the computations leading up to the final FSA will fit in memory.
- **Time efficient**: Likewise, since the word lists created from FSCL descriptions could be quite large, it is important that the algorithm be time efficient so that the computation can be done in a reasonable timeframe.

4.2.1 Previous Work

There are several types of algorithms in the literature which can convert a list of words into a minimized FSA. The two main types are:

- **Post-construction minimization**: Ones which create the FSA in a brute force manner, then minimize it once it has been fully created. These algorithms are time efficient but tend to be space intensive. [37]
- **Incremental minimization**: Ones which create the FSA word by word, minimizing after each word addition. These tend to be more space efficient than the previous type, but require more time. [13]

Since both time and space efficiency are both important to this step, the algorithm that was ultimately settled on for this process was one which fell somewhere between the two extremes. It constructs the automaton incrementally, partially minimizing each step of the way, then performs a final minimization step after the full word list has been added [38]. This achieved a balance between time efficiency and space efficiency that was deemed suitable for the purposes of this project. However, this algorithm required the word list to be in order of decreasing word length. This meant that the word list had to be sorted after being created and before being sent to the FSA creation algorithm.

4.2.2 Description of Algorithms

4.2.2.1 Word List to FSA

Given a word list, this algorithm can convert it into an FSA. To do this, the algorithm creates states and transitions based on the letters in the words it encounters, branching where appropriate and minimizing the automaton as it goes. A precondition of the particular algorithm that was chosen for this process is that the list be sorted in decreasing order of length. The pseudocode for this algorithm is described in Figures 4.4 to 4.6.

```
type FiniteStateAutomaton
   State Array states          // The states in the automaton
   State starting_state        // The starting state
   State Array transition      // A two-dimensional array indexed
                                  by a state and a symbol,
                                  containing a reference to a
                                  state (or null)

global FiniteStateAutomaton fsa  // The automaton
global State Array u             // Set of minimized states
global State Array v             // Set of non-minimized states

Function CreateFSAFromWordList
   ReadWords
   return fsa

Function AddState(State from, String letter)
   create a new state s in fsa
   set transition(from, letter) to s
   return s

Function AddWord(String w)
   q <- fsa.starting_state
   for each letter l in w
      if transition(q, l) = null then
         q <- AddState(q, l)
      else
         q <- transition(q, l)
   q.final <- true
   return q
```

Figure 4.4: Pseudocode of word list to FSA conversion algorithm

```
Function AreEquivalent(State a, State b)
   result <- false
   if a.final and b.final then
      result <- true
      for each letter l in the alphabet
         if transition(a, l) != transition(b, l) then
            result <- false
   else
      result <- false
   return result

Function NextEquivalent(State n)
   result <- null
   for each state s in fsa
      if s != n then
         if AreEquivalent(s, n) then
            result <- s
   return result

Function BuildStack(State q)
   create a new state s in v
   s <- q
   for each letter l in the alphabet
      if transition(q, l) != null then
         r <- transition(q, l)
         if not r.final then
            BuildStack(r)
   return
```

Figure 4.5: Pseudocode of word list to FSA conversion algorithm (cont'd)

```
Function Merge(State a, State b)
   for each state s in fsa
      for each letter l in the alphabet
         if transition(s, l) = b then
            set transition(s, l) to a
   remove b from u
   remove b from v
   return

Function Minimize
   do while v is not empty
      q <- v.pop
      p <- NextEquivalent(q)
      if p != null then
         Merge(p, q)
      else
         add q to u
   return

Function ReadWords
   for each word w in input file
      BuildStack(AddWord(w))
      Minimize
   BuildStack(fsa.starting_state)
   Minimize
   return
```

Figure 4.6: Pseudocode of word list to FSA conversion algorithm (cont'd)

4.2.2.2 FSCL Files to Word List

In order to apply the algorithm in Figures 4.4 to 4.6, the FSCL file must be converted from its compact form into a comprehensive list of words, sorted in decreasing order of length. Therefore, there are two pre-processing steps involved:

1. The word list must be created from the lexicons. To do this, each possible prefix is created by combining the word form of each entry in the initial prefix lexicon with each form in its continuation class that does not cause a clash due to conflicting flag diacritics. The same process is then followed for suffixes, after which each combination of prefix, stem, and suffix that does not cause a flag diacritic clash is produced. The pseudocode for this algorithm is described in Figure 4.7.

2. The constructed list of words must then be sorted in order of decreasing length. To do this, a sorting method is used whereby the algorithm creates a secondary list by iterating through the first list one entry at a time and inserting each entry encountered into the second list before the first entry encountered in the second list with a shorter length. The pseudocode for this algorithm is described in Figure 4.8. Although the algorithm that was implemented uses a form of insertion sort, it was subsequently realized that a bucket sort approach would also have been applicable, and more efficient.

To illustrate this process, the FSCL description given in Figure 4.9 would produce the (unsorted) word list given in Figure 4.10. This would then be sorted into the word list given in Figure 4.11.

```
Function CreateWordList(File prefix, File suffix, File stem)
   result <- {}
   prefixes <- Combine(prefix.lexicons[1])
   suffixes <- Combine(suffix.lexicons[1])
   For each Form f in stem.forms
      For each Form p in prefixes
         For each Form s in suffixes
            new_form <- p + f + s
            if not Clash(new_form)
               result <- result U {new_form}
   return result

Function Combine(Lexicon l)
   result <- {}
   For each Entry e in l
      if e.continuation_class = "#" then
         result <- result U {e.form}
      else
         temp <- Combine(e.continuation_class)
         For each String s in temp
            new_form <- e.form + s
            if not Clash(new_form)
               result <- result U {new_form}
   return result

Function Clash(String s)
   result <- false
   For each FlagDiacritic f in s
      For each FlagDiacritic d in s
         if f.name = d.name and f.value != d.value then
            result <- true
   return result
```

Figure 4.7: Pseudocode for creating an unsorted word list from a FSCL description

```
Function SortByLength(String Array words)
   result <- []
   for each string s in words
      inserted <- false
      for each string r in result
         if s.length > r.length then
            insert s into result before r
            inserted <- true
            exit for
      if not inserted then
         append s to result
   return result
```

Figure 4.8: Pseudocode for sorting the word list

```
LEXICON P
            R;
re          R;
un          R;

LEXICON R
do          S;
wind        S;

LEXICON S
            #;
ing         #;
er          #;
```

Figure 4.9: Example of compiling a FSCL description into a sorted word list

```
do
wind
doing
winding
doer
winder
redo
rewind
redoing
rewinding
redoer
rewinder
undo
unwind
undoing
unwinding
undoer
unwinder
```

Figure 4.10: Example of compiling a FSCL description into a sorted word list (cont'd)

```
rewinding
unwinding
rewinder
unwinder
winding
redoing
undoing
winder
rewind
redoer
unwind
undoer
doing
wind
doer
redo
undo
do
```

Figure 4.11: Example of compiling a FSCL description into a sorted word list (cont'd)

4.2.3 Discussion

```
Function CreateWordList(File input)
   f <- first lexicon in input
   return Merge(f)

Function Merge(Lexicon l)
   result <- {}
   for each entry e in l
      if e.continuation_class = "#" then
         result <- result U {e.form}
      else
         result <- result U Merge(e.continuation_class)
   return result
```

Figure 4.12: Pseudocode for former method of creating a word list from a (single-file) FSCL description

The original implementation of the FSCL interpreter used a single lexicon file similar to LEXC and recursively combined the words based solely on lexicon and continuation class, treating the list of stems as simply another lexicon. The pseudocode for this algorithm is described in Figure 4.12. While at first this algorithm may seem much simpler and elegant than the one ultimately settled on (Figure 4.7), it turns out that in practice it ends up doing a lot of unnecessary and redundant work.

To illustrate, suppose there are three lexicons: A, B, and C. The number of forms contained by these lexicons are a, b, and c respectively. Every entry in A has continuation class B, every entry in B has continuation class C, and every entry in C has continuation class #.

Suppose that there are no flag diacritics in any of the lexicons. In such a case, the recursive algorithm would take time proportional to $a \times b \times c$. The iterative algorithm, however, would either combine A and B into a temporary intermediate lexicon AB, which is then combined with C; or would combine B and C into a temporary intermediate lexicon BC which A would then be combined with. In the first situation, it would first take $a \times b$ time to combine A and B into an intermediate list of $a \times b$ forms, then take $a \times b \times c$ time to combine that list with C to create the final list. Thus the total time taken would be $a \times b + a \times b \times c$. In the second situation, it would take $b \times c$ time to create the intermediate lexicon, and then $a \times b \times c$ to create the final result, resulting in $b \times c + a \times b \times c$ time.

Thus, in the case that there are no flag diacritics, or at least, no flag diacritic clashes, the recursive algorithm is strictly superior. However, if there are a significant number of flag diacritic clashes in the lexicons, the iterative algorithm becomes much faster due to the reduction in size of the intermediate lexicon.

For example, suppose A contained 10 forms, B contains 100, and C contains 30. In that case, the number of combination operations, and thus the time taken by the recursive algorithm would be $10 \times 100 \times 30 = 30,000$, regardless of how many words actually end up being eliminated due to flag diacritic clashes. Now consider the iterative algorithm. Suppose that A and B are combined first, resulting in $10 \times 100 = 1000$ combination operations. However, suppose that due to flag diacritic clashes, only 500 of those combinations are legal. This means the next step will combine AB with C, resulting in $500 \times 30 = 15,000$ combination operations. Thus the total time taken would be only $1000 + 15,000 = 16,000$; far less than the 30,000 taken by the recursive algorithm.

Note, however, that in order for the iterative algorithm to be appropriate, there are several conditions that must be met:

- **The number of flag diacritic clashes must be significant.** If clashes do not significantly reduce the size of the intermediate lexicon, it will take even more time than the recursive algorithm. In general, FSCL files can be expected to meet this condition, since flag diacritics are such a fundamental part of its operation, and, as is not always the case in LEXC, their very purpose is to clash and eliminate illegal word forms. For example, the list of Innu nouns produced from the testing data contained 449,677 words, whereas if no clash removal had been performed, roughly 2,949,625,570,500 words would have been produced.

- **Flag diacritic values must not be changeable.** In LEXC, it is possible for a particular flag diacritic's value to be changed or cleared during the combination process. In such a case, a clash in the intermediate lexicon would not be sufficient grounds to eliminate the partial word from it, since combining it with a subsequent word form may undo the clash. In FSCL, however, the values of flag diacritics are immutable during the combination process and thus a clash in a partial word *is* sufficient grounds to eliminate it from the list since adding further word forms cannot undo the clash.

- **The continuation class of each entry in each lexicon being combined this way must be the next lexicon in the sequence.** It would be meaningless to combine A with B if some entries in A had continuation class B, some had C, and some had #. In LEXC, there is no restriction on what lexicon any given entry can have as its continuation class. However, FSCL has been designed around the idea that any given word may or may not have some prefixes, and may or may not have some suffixes, but must have a stem. Thus, it is safe to divide all lexicons into those three categories (prefix, stem, and suffix), and have a separate file for each category. Thus, there may be several prefix lexicons, or there may be none, but definitely whatever legal prefixes there are must come before anything in the stem file. Likewise, there may be several suffix lexicons, or there may be none, but definitely whatever legal suffixes there are must come after anything in the stem file. The stem file may contain several lexicons which combine together to form stems, or may be one monolithic lexicon, but it cannot be empty.

Since FSCL meets all the required criteria, the iterative algorithm is more appropriate.

Let n represent the number of entries in the lexicon with the most entries in the prefix file. Let s represent the number of entries in the lexicon with the most entries in the stem file. Let m represent the number of entries in the lexicon with the most entries in the suffix file. Let k represent the number of lexicons in the prefix file, l the number of lexicons in the stem file, and p the number of lexicons in the suffix file. Since each entry in a given lexicon can only take a lexicon that comes after it as a continuation class, the worst case would be if every entry's continuation class in every lexicon was the immediately subsequent lexicon. This means the total number of prefixes would be n^k, the total number of stems would be s^l, and the total number of suffixes would be m^p. However, the tree-like nature of the recursion means twice as many steps would be involved in constructing those word forms. Thus, the worst-case running time would be roughly on the order of $2n^k \times 2s^l \times 2m^p$. While this result may seem intimidating, it is important to point out that it makes the assumption that no flag diacritic clashes occur. However, it turns out that a high rate of flag diacritic clashing results in a high reduction of running time. Thus, in practice, the running time is far below that of the worst case. Note that the running time is directly related to the number of forms being combined, and therefore, measuring the number of forms eliminated can give a rough idea of the amount of running time saved. In the actual data used to perform the testing of FSCL on Innu nouns, there were relatively few prefixes to be combined, and the stems were all stored in a single lexicon, however the suffixes would have generated roughly 941,000 forms by the recursive algorithm. After eliminating clashes, however, it resulted in a mere 186 forms.

Now that the implementation details have been described, the next chapter will describe the testing process, as well as the results and conclusions it produced.

Chapter 5

Conclusions

Now that the FSCL specification language has been described, as well as a system capable of interpreting and converting FSCL descriptions into FSAs and performing spelling correction on those FSAs, the system can be tested to determine the suitability of FSCL and its associated tools for its intended target natural languages. To do this, part of the Innu language will be encoded in FSCL. Section 5.1 a description of the results of testing the FSCL system on the Innu language. Then, Section 5.2 discusses possibilities for expanding on this work in the future. Finally, Section 5.3 discusses what has been learned and demonstrated in this thesis.

5.1 Results

Once the specification language itself has been defined, and a software system capable of interpreting that language and performing spelling correction operations on the FSAs output by the interpreter has been implemented, the language can then be tested to determine whether it indeed meets its intended design criteria. FSCL was thus tested by attempting to encode the Innu language. Innu is an agglutinative aboriginal language and hence would be an appropriate natural language on which to test FSCL's encoding capabilities and the performance efficiency of the algorithms implemented to operate on it.

First, Section 5.1.1 will discuss some features of the Innu language and how those features motivated decisions regarding the nature of the testing data. Then, Section 5.1.2 will describe the process used to test the FSCL system using that data. Finally, Section 5.1.3 will report the results of the testing.

5.1.1 Innu Language Description

Innu is a highly agglutinative language, which, as discussed in Section 2.1.2 means that many ideas which in most other languages would be expressed with separate words are instead expressed with affixes to one of the main words of the sentence. See Figure 5.1 for an example of this phenomenon.

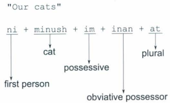


Figure 5.1: Example of agglutination in Innu.

The Innu language consists of three primary parts of speech [11]:

- **Particles**: These are words which represent abstract grammatical concepts and connections, such as conjunctions and prepositions. Particles in Innu have fixed forms which do not accept affixes, thus making them trivial to encode.
- **Nouns**: These are words which represent people, places, and things. Nouns in Innu have a more complicated structure than particles, and accept many types of affixes, thus making them more difficult to encode.
- **Verbs**: In many languages, verbs often represent actions. Innu verbs, however, can include concepts that in other languages would be categorized differently, such as the property of being red or being daytime. Verbs in Innu are much more complicated than nouns, making them much more difficult to encode.

It was decided that it would be sufficient for the testing of the FSCL system to encode only the members of one of the three main parts of speech for the purposes of testing the FSCL specification language; specifically, the nouns. Particles allow no affixation, and thus would have been too trivial to fully test the limits of the system. The structure of Innu verbs is notoriously complex; but the nature of this complexity is such that it does not bring anything fundamentally new to the table which cannot be tested with the simpler structure of Innu nouns. Therefore only the nouns were implemented to test the system.

5.1.2 Description of Testing Process

The first step in testing was to use some simple but comprehensive dummy data to test the accuracy of the implementation of the integrated FSCL system described in the previous chapter. Once its accuracy had been verified, the capabilities of the FSCL language were fully tested using Innu, by way of the following process:

1. **Obtain language data from linguists:** As the author is not an expert in Innu, some knowledgeable consultants were approached, including Marguerite Mackenzie, a prominent scholar of the Innu language. Two types of relevant information were obtained from these consultants. The first was a description of the structure of Innu nouns and how they are formed, as well as a comprehensive list of possible word stems (roughly 11500 stems) and affixes (roughly 120 affixes); the second was a list of both correctly spelled words and incorrectly spelled words for the purposes of accuracy verification (roughly 120 words).

2. **Encode data in FSCL:** Once the relevant information had been obtained, the stems, affixes, and processes by which they are combined, an initial FSCL encoding was created.

3. **Compile and test:** Once that data had been encoded, the FSCL description was transformed into an FSA which was given as input to the spelling correction algorithm, by the processes discussed in the previous chapter. The list of testing words obtained in Step 1 above were then fed into the spelling corrector and the resulting output was recorded.

4. **Verify results:** The results recorded in the previous step were then shown to the linguist consultants who analyzed them to look for instances of correct words mistakenly corrected into incorrect words, or incorrect words mistakenly recognized as correct words. They then corrected any misunderstandings or misinterpretations the author had with regard to the language information provided in Step 1 above that may have led to the observed errors.

5. **Revise and test:** With the new information and corrections from the linguist consultants, the author then revised the FSCL encoding and proceeded again with Step 3 above, repeating this process as deemed necessary.

5.1.3 Discussion

The process of encoding the Innu language in FSCL was an iterative one, rather comparable to the process of soliciting software requirements in software engineering. Language data was gathered from the linguists, implemented as best understood by the author, then the results examined and verified by the linguists, and corrections made. This process was repeated until no further corrections were deemed necessary (in this case, three times).

One should expect any language encoding exercise to have this nature. Even if the person doing the encoding is a language expert himself or herself, he or she must not expect the encoding to necessarily be completely error-free the first time. The encoding must be rigorously tested and debugged.

Even after iterating this process several times, a few small errors remained in the output (roughly 7 mistakes out of 117 test words). However, these are a result of mistakes in the details of the particular encoding, arising from the author's inexperience with the Innu language, and not a result of FSCL's inability to code fundamental language features. Should someone more fluent in Innu create a more comprehensive encoding, FSCL has sufficient expressive power to correctly encode it.

5.2 Future Work

There are many ways this work could be built upon in the future, such as the implementation of other spelling correction features like phonetic correction or context-sensitive correction. However, this section will primarily focus on two main ideas: extending the existing dictionary for Innu to encode the rest of the language (or even other similar languages), and implementing a better user interface for the spelling correction utility.

5.2.1 Extending the Dictionary

5.2.1.1 Completing the Dictionary

Innu verbs are far richer and have a much more complicated structure than Innu nouns. One way in which the limits of FSCL can truly be tested would be to complete the encoding of the Innu language by including the verb dictionary.

In addition to testing FSCL's capabilities on more complicated structures, doing this would provide the Innu community with a complete spelling correction software resource which will help to preserve their language.

5.2.1.2 Creating other Dictionaries

Though FSCL was only tested on the Innu language, there are other northern North American aboriginal languages in need of computational language resources such as spelling correctors. Creating such software by encoding those languages in FSCL would be far simpler and easier to modify than it would be if it were done in comparable software, and would be a great help to the communities to which those languages belong.

5.2.2 User Interface

Though the spelling correction software that was currently written for use with the automata created from FSCL descriptions is correct and efficient, it does not currently have much in the way of a user interface. The user must type a single word at a terminal command prompt after which the program will compute and display a list of corrections. A proper interface should be able to interact with documents in standard formats, scanning each word in the document for errors, and allowing the user to select words from the provided list of suggested corrections to replace the error word.

There are some issues, however, that need to be addressed if one is to create a proper user interface:

- **Error threshold**: What would be an appropriate error distance for the algorithm to scan out to? Too small a distance could cause many words to have empty correction lists, which would not be very helpful to the user. Likewise, too large a distance could cause many words to have an unreasonably long correction list which would also not be very helpful to the user since he or she would have to spend a great deal of time sifting through the possibilities to find the one he or she is looking for (which still may not be there).

One possibility is that the interface designer could take advantage of the nature of the error distance concept. As the list of corrections provided at distance d is always a subset of the corrections provided at distance $d + 1$, the interface could be designed in such a way that, for example, the default error distance for the spell checker is low, but should the user not find the correct word in the short list provided at that distance, he or she can expand the list by having the algorithm scan the word again at the next distance out. The user may then repeat this process until the correct word is found.

- **High misspelling rates**: Since the aboriginal languages for which FSCL was designed typically have only recently developed a writing system and have relatively low literacy rates, documents written in these languages tend to have high rates of misspelling. That is, a significant proportion of the words in the documents contain at least one error. This means that using spelling correction software on a large or even moderate length document in these languages may be a time-consuming task.

  To alleviate this problem somewhat, the user interface may be designed in a way that facilitates the rapid or automatic correction of commonly misspelled words. There are existing techniques available that other spelling correction software has implemented that could be used for this purpose. For example, one such technique takes a list of common misspellings and their corrections associated with the language (in much the same way as the replace rule technique used by XFST), and before running the comprehensive spelling correction system the user can simply have the document "auto-correct", replacing all instances of misspellings in the list replaced by their corresponding corrections.

5.3 Summary

FSCL is well-suited to encoding agglutinative natural languages. The lexicon and continuation class system it uses facilitates agglutination, while its small set of meta-information specification features generally simplifies the encoding of languages.

FSCL works well with finite state spelling correction. The interpreter can easily convert a FSCL description into an FSA, and the depth-first search with error tolerance algorithm which operates on that FSA allows for efficient correction of long words with few errors per word, even with large FSAs.

Finally, the testing done with FSCL has demonstrated its ability to do its job effectively, given that the natural language in question has been accurately encoded.

Bibliography

[1] Akmajian, A. (2010) *Linguistics: An Introduction to Language and Communication.* (6th Edition). MIT Press.

[2] Alanen M. and Porres I. (2003) *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels.* Turku Centre for Computer Science. Turku, Finland.

[3] Alegria, I., Aranzabe, M., Ezeiza, N., Ezeiza, A., and Urizar, R. (2001) "Using Finite State Technology in Natural Language Processing of Basque." In *Conference on Implementation and Application of Automata 2001.* Lecture Notes in Computer Science no. 2494. Springer-Verlag. 1-12.

[4] Backus, J. (1959) "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference." *Proceedings of the International Conference on Information Processing.* UNESCO. 125–132.

[5] Baraby, A. (2002) "The Process of Spelling Standardization of Innu-aimun (Montagnais)." *Indigenous Languages across the Community.* North Arizona University, Flagstaff.
`http://jan.ucc.nau.edu/ jar/ILAC/ILAC_21.pdf`

[6] Bartlett, S., Kondrak, G., and Cherry, C. (2008) "Automatic Syllabification with Structured SVMs for Letter-To-Phoneme Conversion." *Proceedings of Association for Computational Linguistics 2008*. Association for Computational Linguistics. 568–576

[7] Beesley, K.R. and Karttunen, L. (2003) *Finite-State Morphology*. Center for the Study of Language and Information Publications; Stanford, CA.

[8] Brill, E. (1992) "A Simple Rule-Based Part-of-Speech Tagger." In *Proceedings of the Third Conference on Applied Natural Language Processing* Association for Computational Linguistics. Trento, Italy. 152–155

[9] Bruggemann-Klein, A. (1993) "Regular Expressions into Finite Automata." *Theoretical Computer Science* 120, 197–213.

[10] Burnaby, B. (2004) "Linguistic & Cultural Evolution in an Unyielding Environment" in *Cultural Diversity and Education: Interface Issues*. Memorial University, St. John's. 31–50.

[11] Clarke, S. (1982) *North-West River (Sheshatshit) Montagnais: A Grammatical Sketch* National Museums of Canada, National Museum of Man, Mercury Series, Canadian Ethnology Service Paper No. 80; Ottawa, Canada.

[12] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001) *Introduction to Algorithms* MIT Press and McGraw-Hill.

[13] Daciuk, J., Watson, B., and Watson, R. (1998) "Incremental Construction of Minimal Acyclic Finite State Automata and Transducers." *Proceedings of the*

International Workshop on Finite State Methods in Natural Language Processing Association for Computational Linguistics. Ankara, Turkey. 48–56.

[14] Dale, R., Moisl, H., and Somers, H. (2000) *Handbook of Natural Language Processing*. Marcel Dekker. New York.

[15] Gries, S. "Shouldn't it be a *breakfunch*? A Quantitative analysis of blend structure in English." *Linguistics* 42, 639-67

[16] Karttunen, L. (2001) "Applications of Finite-State Transducers in Natural Langauge Processing." *Proceedings of the 5th International Conference on Implementation and Application of Automata*. Lecture Notes in Computer Science no. 2088. Springer. Berlin. 34–46

[17] Kukich, K. (1992) "Techniques for Automatically Correcting Words in Text." *ACM Computing Surveys*, 24(4), 377–439.

[18] Linz, P. (2006) *An Introduction to Formal Languages and Automata* (4th Edition). Jones and Bartlett Publishers.

[19] Mackenzie, M. (2006) Personal communciation.

[20] McDonough, J., Whalen, D. (2008) "The Phonetics of Native North American Languages." *Journal of Phonetics*, 36(3), 423–426

[21] McNamee, P. and Mayfield, J. (2004) "Character N-Gram Tokenization for European Language Text Retrieval." *Information Retrieval*, 7, 73–97.

[22] Mihov, S. and Schulz, K.U.. (2004) "Fast Approximate Search in Large Dictionaries." *Computational Linguistics*, 30(4), 451-477.

[23] Myers, E.W. and Miller, W. (1989) "Approximate Matching of Regular Expressions." *Bulletin of Mathematical Biology*, 51, 5-37.

[24] Oflazer, K. (1996) "Error-Tolerant Finite-State Recognition with Applications to Morphological Analysis and Spelling Correction." *Computational Linguistics*, 22(1), 73-89.

[25] Pagh, R. (2001) "Low Redundancy in Static Dictionaries with Constant Query Time." *SIAM Journal of Computing*, 31(2), 353-363

[26] Parkes, A. (2008) *A Concise Introduction to Languages and Machines* Springer. New York.

[27] Radford, A., Atkinson, M., Britan, D., Clahsen, H., and Spencer, A. (2009) *Linguistics: An Introduction* (2nd Edition). Cambridge University Press.

[28] Roche, E. and Schabes, Y. (1995) "Deterministic Part-of-Speech Tagging with Finite-State Transducers" *Computational Linguistics*, 21(2), 227–253

[29] Roche, E. and Shabes, Y. (1997) *Finite-State Language Processing.* MIT Press, Cambridge.

[30] Savary, A. (2002) "Typographical Nearest-Neighbor Search in a Finite-State Lexicon and Its Application to Spelling Correction." In *Conference on Implementation and Application of Automata 2002.* Lecture Notes in Computer Science no. 2494. Springer-Verlag; Berlin. 251–260.

[31] Van Valin, R. and LaPolla, R. (1997) *Syntax: Structure, Meaning, and Function* Cambridge University Press.

[32] Vilares, M., Otero, J., Barcala, F.M., and Dominguez, J. (2004) "Automatic Spelling Correction in Galician." In *EsTAL 2004*. Lecture Notes in Artificial Intelligence no. 3230. Springer-Verlag; Berlin. 45–57.

[33] Vilares, M., Otero, J., Grana, J. (2004) "Regional Finite-State Error Repair." In *Proceedings of the Conference on Implementation and Application of Automata 2004*. Lecture Notes in Computer Science no. 3317. Springer-Verlag; Berlin. 269–280.

[34] Vilares, M., Otero, J., and Grana, J. (2005) "Regional Versus Global Finite-State Error Repair." In *Proceedings of the Conference on Intelligent Text Processing and Computational Linguistics 2005*. Lecture Notes in Computer Science no. 3406. Springer-Verlag; Berlin. 120–131.

[35] Vilares, M., Otero, J., and Vilares, J. (2006) "Robust Spelling Correction." In *Proceedings of the Conference on Implementation and Application of Automata 2005*. Lecture Notes in Computer Science no. 3406. Springer-Verlag; Berlin. 120–131.

[36] Wagner, R.A. (1974) "Order-n Correction for Regular Languages." *Communications of the ACM*, 17(5), 265–268.

[37] Watson, B. (2002) "A Fast and Simple Algorithm for Constructing Minimal Acyclic Deterministic Finite Automata." *Journal of Computer Science*, 8(2), 363–367

[38] Watson, B. (2003) "A New Algorithm for the Construction of Minimal Acyclic DFAs." *Science of Computer Programming*, 48, 81–97.

[39] Zerrouki, T., Palla A. (2009) "Implementation of Infixes and Circumfixes in the Spellcheckers" *Proceedings of the Second International Conference on Arabic Language Resources and Tools.* The MEDAR Consortium; Cairo, Egypt. 61–65.

Appendix A

Test Data

Word	False Negative
`uatikw`	
`mishtikw`	
`Nishunnu`	
`nitassinan`	*
`nitauassim`	*
`anite`	
`mishikamau`	*
`pitukamit`	*
`uapikun`	
`akaneshau`	
`uapikun-pishimw`	

Figure A.1: The list of correctly spelled Innu words used for testing. Those with an asterisk in the column labelled "False Negative" are ones which the spelling corrector mistook as being incorrectly spelled.

Word	False Negative
massin	
minekash	
mashten	
atusseun	
atikw	
mashinaikan	
anitshenat	
aimu	
eukuan	
mukw	
Innuat	

Figure A.2: The list of correctly spelled Innu words used for testing (cont'd).

Word	False Positive
uatuk	
uatukw	
uatuku	
uataku	
uatek	
uaatik	
uateku	
uatik	
uatakw	
mistuk	
mistuku	
mishtuk	
mishtuku	
mishtukw	
mistikw	
mistiku	
nishunu	
nishinu	
nishinnu	
nissinnu	
nissinu	
niishunnu	
ntassinan	
ntasinan	
ntaasinan	
ntasiinan	
nitassinaan	
nitasinan	

Figure A.3: The list of incorrectly spelled Innu words used for testing. Those with an asterisk in the column labelled "False Positive" are ones which the spelling corrector mistook as being correctly spelled.

Word	False Positive
ntauassim	
nitauasim	
ntauasim	
ante	
nte	
nite	*
nate	
mishakamau	
mishekamau	
missekamau	
misekamau	
missakamau	
misakamau	
pitakamit	*
pitekamit	
piitukamit	
pitukamiit	
pupun	
uapakun	
uapekun	
uaapikun	
uapikuun	
uapukun	
akanneshau	
akeneshau	
Akaneshau	

Figure A.4: The list of incorrectly spelled Innu words used for testing (cont'd).

Word	False Positive
uapikunpishimw	
uapikun pishimw	*
uapakun-pishimw	
uapikun-pishumw	
Uapikun-pishimw	
Uapakun-pishimw	
Uapakun-pishumw	
uapikun-pihumw	
uapikun-pihimw	
masin	
maassin	
masen	
masan	
minekass	
miinekash	
miinekas	
masten	
massten	
mastan	
mashtan	
maashten	
maasten	
mashtenn	
atuseun	
attuseun	
attusseun	

Figure A.5: The list of incorrectly spelled Innu words used for testing (cont'd).

Word	False Positive
atik	
attik	
attikw	
attiku	
aatik	
atiik	
mashineikan	
ntshent	
ntshe	
antshent	
antshenat	
eimu	
eukun	
muku	
mukw	
Innut	
Inut	
Inuat	

Figure A.6: The list of incorrectly spelled Innu words used for testing (cont'd).