Karlsruhe Institute of Technology

Carnegie Mellon University

A System for Recognizing Natural Spelling of English Words

Diploma Thesis
by

Lucas Czech

at the
Interactive Systems Labs (ISL)

Karlsruhe Institute of Technology (KIT),
Karlsruhe, Germany

Carnegie Mellon University (CMU),
Pittsburgh, USA

First Reviewer:	Prof. Dr. Alexander Waibel
Second Reviewer:	Dr. Sebastian Stüker
Advisor:	Dipl.-Inform. Thilo Köhler

May 7, 2014

Zusammenfassung

Buchstabieren ist ein nützlicher Weg, um eine exakte Buchstabenfolge zu übermitteln. Wenn zwei Personen einander ein Wort buchstabieren, können sie die Buchstaben um zusätzliche Codewörter nach dem Schema „A wie Anton“ erweitern, um Missverständnisse zu vermeiden. In dieser Arbeit wird ein Spracherkennungssystem vorgestellt, dass mit solchen Phrasen umgehen kann, die zum Buchstabieren im Englischen genutzt werden. Dies schließt auch Ausdrücke für Groß-/Kleinschreibung, Zahlen und Leerzeichen mit ein. Das System extrahiert die Informationen aus einer Sprachäußerung und leitet das darin buchstabierte Wort ab.

Es gibt viele Anwendungsfälle für ein solches System, unter anderem das Lernen von Out-of-Vocabulary Wörtern, Fehlerbehebung und Dialogsysteme, wie zum Beispiel Warenlager oder Flugbuchungen. Der Erkenner besteht aus zwei Hauptkomponenten, die auf dem Janus Spracherkennungs Toolkit beruhen. Die erste ist ein Dekodierer, der ein auf Buchstabieren spezialisiertes Sprachmodell nutzt, welches auf einem geeigneten Textkorpus trainiert wurde. Sie gibt ein Confusion Network aus, das buchstabierte Phrasen enthält. Die zweite Komponente ist ein Parser, der über eine Tiefensuche mit Pruning durch einen Strahl das Confusion Network durchsucht. Sie leitet dann die Folge der Buchstaben aus den Phrasen, die im Network gefunden werden, ab.

Es werden außerdem zwei Verbesserungsmethoden für das System vorgestellt. Eine geht das Problem an, dass viele Buchstaben im Englischen gleichklingend mit Wörtern sind (zum Beispiel ‚S‘ und „as“), während die andere Diskrepanzen auflöst, die beim Dekodieren entstehen können, wie zum Beispiel „B as in pepper“. Außerdem wird ein System mit Live-Feedback präsentiert, das unmittelbar Fehler einzelner Buchstaben behebt, indem der Nutzer aufgefordert wird, diesen Buchstaben zu wiederholen.

Das System wird anhand von Audiodaten evaluiert, die zum Testen aufgenommen wurden, und es wird mit zwei für diesem Zweck entwickelten Baseline-Systemen verglichen: Einem, das spezialisiert ist auf die Erkennung von Einzelbuchstaben und einem, das auf einem Sprachmodell mit kontextfreier Grammatik beruht. Das System erreicht ähnliche Ergebnisse wie das erstgenannte Baseline-System auf einer geeigneten Untermenge der Daten, die auf Einzelbuchstaben eingeschränkt ist, und übertrifft das zweite Baseline-System sowohl in Genauigkeit als auch Geschwindigkeit.

Abstract

Spelling is a useful way of communicating an exact sequence of letters. When people spell a word to one another, they tend to elaborate on the letters being spelled by using additional codewords like “A as in alpha” in order to avoid misunderstandings. This work presents a speech recognition system for the kinds of phrases that are used for spelling in English. This also includes expressions for capitalization, numbers and spaces. It extracts the information from a speech utterance and infers the word being spelled.

There are plenty of use cases for such a system, including out-of-vocabulary learning, error recovery and dialog systems, for example in warehousing or flight booking. The recognizer consists of two main components which are based on the Janus recognition toolkit. The first one is a decoder, which uses a specialized language model for spelling that is trained on a suitable text corpus. It outputs a confusion network containing the spelling phrases. The second component is a parser using a depths first search with pruning via a beam to search the confusion network. It then infers the sequence of spelled letters from the phrases it finds in the network.

This work also presents two improvement methods for the system. One that addresses problems arising from the homophony of most English letters with a word (for example, ‘S’ and “as”) and another one that resolves mismatches occurring in decoding like “B as in pepper”. Furthermore, a live feedback variant of the system is presented that immediately corrects single letter errors by having the user rephrase that particular letter.

The system is evaluated on audio data that was recorded for testing, and is compared to two specially developed baseline systems, one specialized on single letters and another that is based on a context-free grammar language model. The system achieves results comparable to the former on a data subset consisting only of single letters and outperforms the latter in both accuracy and speed.

I hereby declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, May 7, 2014

. . . . . .

(**Lucas Czech**)

Acknowledgment

There are many people who helped and supported me in different ways during the conduction of this thesis.

First, I would like to thank my advisor Professor Dr. Alexander Waibel, who not only gave the incentive for this research project and accompanied its course, but also gave me the opportunity to conduct the development, data collection and experiments at Carnegie Mellon University in Pittsburgh, USA.

Many thanks also to my second advisor Dr. Sebastian Stüker for his support and advice. My special thanks go to Thilo Köhler, who had wide influence on the methods used in this work and their implementation, and to Tim Notari, without whom the data collection would not have been possible that was necessary to evaluate the systems.

Also, I want to thank Dr. Christian Fügen, Dr. Florian Metze, Kevin Kilgour, Ian Lane, and the other people at the Interactive Systems Lab and at Mobile Technologies in both Pittsburgh and Karlsruhe for their support whenever I needed it.

Further my thanks go to Faith Boldt, Margit Rödder and Silke Dannenmaier for helping me organizing the exchange to CMU, and of course I want to thank the interACT program and the Baden-Württemberg Stiftung for supporting my research stay.

To the team at the Lab in Pittsburgh — namely Christina Agnes Milo, Andreas Veit, Georg Binder, Anuj Kumar, Anna Donohoe and Susanne Burger — also many thanks, you made my stay even more enjoyable.

As for the data collection, I want to thank all the unnamed people taking part in it and providing their voice for science.

Finally, I want to thank my parents Peter and Maria and my sister Judith, who not only constantly supported me during this thesis, but through all of my years of study in Karlsruhe and around the world.

interACT

Baden-Württemberg STIPENDIUM

Ein Programm der Baden-Württemberg Stiftung

WIR STIFTEN ZUKUNFT

Contents

List of Figures

List of Tables

1. Introduction

In this chapter, we first provide some motivation for the topic of this work and present its objective and contribution. Then, the structure of the subsequent chapters is declared and a review of related work is given.

1.1 Motivation

In the beginnings of speech recognition, single letters and sequences of letters were the first tasks to be approached, mainly because of limited knowledge and resources of that time. Later, systems for recognizing continuous speech were developed, but letter sequences were still used as a means of error correction and inputting out-of-vocabulary words. In both cases, mostly spelling in form of pronouncing the names of the letters was used.

However, when people have to spell a word to one another instead of a computer, they tend to use more than just single letters to communicate a spelled word, particularly when the correct transfer of the exact sequence is necessary. An example that motivated this work was a message left on an answering machine which included the following sentence:

> "My name is Kallmeter. K as in kilo, A, double L, and then meter like the measure."

Unlike the utterances that typical spelled speech recognizers can handle, this one contained not only single letters, but also an elaboration on a letter in form of a codeword "K as in kilo", the expression "double L" and finally the analogy "meter like the measure". Because of the noisy channel of a telephone, it contains all these redundancies and acoustic assistances, which are meant to help the human receiver understand the utterance.

But they might also be helpful for a computer system that is trying to recognize spelled words. As people tend to use such phrasings, computers should adapt and try to make use of them. In this work, we describe a system that detects such

phrases and infers the spelled word from them. Merely the use of analogies was not included, because this implies a deeper understanding of the relationship of the used words (i. e. , "meter" is a unit of measurement). These kinds of semantic analyses are a research topic of its own and not covered in this work.

There are two main design goals for our system: First, it has to orient itself towards the actual way people spell complex words, and second, in doing so it has to provide better accuracy than a system based only on single letters. Such a system has many use cases and can be embedded into another system as a special feature that will be activated on occasions where spelling is necessary or helpful.

1.2 Objective and Contribution

The objective of this work is as follows.

> Given an utterance in audio form containing spelled information as people spell with one another, we want to extract this information and infer the sequence of spelled letters.

The utterance may not only include codewords for the single letters ("A as in alpha"), but also contain other phrases used in spelling, for example to state capitalization of letters, spaces, numbers and so on.

To our knowledge, a system that recognizes these arbitrary sequences of letters using human-like spelling has not yet been proposed. This work presents such a system. Additionally, audio data was collected for testing the system on sequences of naturally spelled words.

1.3 Structure

This work is structured as follows. The remainder of this chapter presents related work of our topic. In Chapter 2, the foundations of spelling as well as speech recognition are introduced. In Chapter 3 follows a description of the preliminary system development, which includes an analysis of the problem and two baseline systems used to evaluate the main system presented here. This system is presented in Chapter 4 and is called phrase network system. In Chapter 5, the collection of audio data used for training and testing the systems is described. Then, an evaluation of the systems is presented in Chapter 6, before a final discussion in Chapter 7 concludes this work.

1.4 Related Work

There are many tasks related to our topic which have been researched before. This section gives an overview of the different fields of research and their approaches in the course of time.

Two publications that use ideas similar to our work are discussed at the end of Section 1.4.3.

1.4.1 Spoken Letter Recognition

The recognition of isolated words in a limited vocabulary was one of the first tasks automatic speech recognition (ASR) was concerned with. Standard vocabularies include the 10 digits (zero to nine), the 26 letters of the English alphabet (A to Z) and command sets for different purposes (e. g. , "stop", "repeat", "enter"). On the one hand, the letters of the alphabet are limited, which leads to a low number of classes; on the other hand, they sound partially similar, which causes classes to be close together in the acoustic feature space. Therefore, various approaches to the problem were tried, exploiting different characteristics of the used algorithms.

Early work in the recognition of spoken letters used template matching with dynamic time warping to compare an input utterance to reference templates on a frame-by-frame matching basis, which is reviewed in [RaWi87] and [CoSL90]. Eventually, feature-based methods were introduced using statistical pattern classification on features extracted from the utterance. [CSPB+83] presents the FEATURE system, which combines expert-knowledge-based features with multivariate classifiers and is an early speaker-independent system for isolated letters.

Later approaches used neural networks for both segmentation and classification of single letters. This approach has the advantage to learn significant characteristics of letters automatically, which is particularly useful for similarly sounding letters like the E-set (see Section 2.1.1). Exemplary systems use fully connected feed-forward networks with one hidden layer and 26 output neurons for the letters, first with isolated letters [CoFa90], then using strings of spoken letters [CFMG90]. A similar system is presented in [CFGJ91] for speaker-independent recognition of 50,000 spelled names, which was later refined to also work with the additional difficulty of using telephone speech instead of clean studio recordings in [FaCR92]. The latter work also reports about their data collection procedure, which is similar to ours as described in Section 5.2.

Extended methods for neural networks are for example presented in [HiWa93], where multi-state time delay neural networks are used to recognize connected letters; this again was later refined to work with telephone speech [HiWa96]. Also, different language models like bigrams, search in an n-best list and fully constrained search to a limited vocabulary were examined for the use with neural networks, as presented in [BeHi95].

In modern ASR systems, Hidden Markov Models (HMM) are commonly used for the acoustically modeling of words and phonemes (see Section 2.2.1). An early whole-word HMM recognizer with continuous density is described in [RaWi87], a similar system tested on letters and digits is [EJLS90]. Later, this was extended to connected letters with no pauses between the letters, as in [JLMG93]. Instead of whole-word HMMs, then context-dependent phoneme-based HMMs were used; for example [LoSp94] presents a system for single letters, which focuses on modeling the subtleties of the E-set, and was trained and evaluated on the ISOLET database [CoMF90]. Furthermore, [LoSp96] presents a high-performance alphabet recognition system with context-dependent phoneme-based HMMs, which goes into detail about specific pairs of letters with high confusability. It proposes special features and models for improving discrimination of some letters like nasal consonants. The issue of similar sounding letter names is discussed in Section 4.5.2.

Also, systems for spelling in other languages than English were built covering special characteristics of the desired alphabet; [RoRM97] is such a system for Portuguese where diacritics are allowed to be added to corresponding letters. As our system concentrates on English, we did not need such extensions; however they are easily implemented on demand, as mentioned in Section 3.2.

As HMMs became more common in ASR, there was also some work on special issues concerning letter recognition. [ThRK00] use automatically determined letter groups and their trigrams as multi-letter units to increase context-length for a spelling system, which is a useful adaption to be used on limited domains. [MiSe99] presents a speedup technique using a fast lexical match for selecting a spelled name from a very large list by mapping the 26 letter of the English alphabet onto 8 classes of confusable letters. We implemented a similar idea of creating a distance between the names of two letters in Section 4.5.2.

In addition, hybrids of neural network and HMM based systems were tested, for example [JVFM95], which uses a multi-pass strategy, where an n-best list is propagated through four modules specialized on different tasks: an acoustic Hidden Markov Model of the letters, a selectively trained neural network, a dynamic time warping alignment step, and a dynamic grammar are applied consecutively.

In later years, ASR research began focusing on continuous speech instead of letters. On the one hand, this means that there are rarely news on letter recognition; on the other hand this gave rise to the basic idea of our recognizer, which additionally to single letters allows words as codes for letters, as well as complete phrases of spelling, and thus is only possible because todays systems are capable of continuous speech recognition.

1.4.2 Formal Languages

This work uses formal languages and grammars like context-free grammars (CFG) and probabilistic context-free grammars (PCFG) for many modeling tasks, for example in Section 3.3.2 and Section 4.3. The topic of formal grammars and CFGs is introduced in Section 2.2.3.

In this section, we present some literature on approaches to infer grammars from text corpora. This is a useful task when a system is intended to use a formal language, but its exact nature is unknown in terms of which production rules and nonterminals are needed. This was not necessary in our work, as after the collection of appropriate data in Section 5.2, it was possible to explicitly model the rules of the grammar, as described in Section 3.2.

The problem when automatically inferring a language from sparse data is that there is a risk of creating a trivial grammar, where each utterance is captured in one production rule. Furthermore, fixed hand-written rules are easily parsable, because they are tailored to the needs of the system, in our case spelling. For these reasons, we modeled our grammar by hand; as it however might be an interesting topic for further work, we present some approaches for automatically inferring rules.

The publications presented here use both regular and context-free grammars. The task of spelling is linear in the form that generally one letter follows another without dependencies, so that a regular grammar is sufficient to model this. However,

for more sophisticated applications, a CFG is necessary. Furthermore, as a suitable component for context-free grammars was already available in our used speech recognition system Janus (see Section 2.3 for an introduction), we used this instead, see Chapter 3.

[Lee96] gives a survey of the literature concerning the topic of learning CFGs in general. [CiKr03] also reviews algorithms for inducing grammars, with a special focus on sparse data sets. Both engage in learning the production rules from a given text corpus.

In order to also learn probabilities for these rules when being used in probabilistic languages, [CaOn94] presents methods to induce these for stochastic regular grammars. For the estimation of stochastic context-free grammars, [LaYo90] uses the Inside-Outside algorithm.

Lastly, an algorithm for inferring a PCFG from sparse data is presented in [JoGG07]. It tries to solve the problem of creating trivial grammars, which the previously mentioned algorithms commonly produce when not provided with enough data.

1.4.3 Named Entities and Confusion Networks

A major part of this work engages in identifying phrases of spelling in an utterance; this task relates to natural language processing, information extraction and named entity retrieval.

The broad field of information extraction from text has many applications and is thus well researched. Often, the objective is named entity detection, e. g. people names, company names or locations have to be found in the input data. [MaSc94] is an early work showing the importance and the difficulties of recognizing proper names (particularly of people) correctly when being uttered or spelled.

[Gris97] gives an overview of early techniques for the general issue of information extraction from a text corpus. In [Gris98], the author later applies this to cover special challenges when using speech recognition, which is also examined in [KSSW98].

Later approaches for information extraction from ASR mostly use confusion networks, which are introduced in Section 2.2.4 as a compact format of representing alternative word sequences resulting form the speech recognizer. In [TWGR+02], confusion networks are used instead of the 1-best result from ASR for the general task of spoken language understanding. This is done by interpreting the confusion network as a finite state machine and matching it with a regular grammar.

An important issue in natural language processing and speech recognition is keyword spotting, which is useful for indexing, archiving and searching utterances. This is related to our task because the objective of keyword spotting is finding short snippets in the input data that contain keywords and phrases. This can be done using lattices [SaSp04] or confusion networks [HHHG07]; the latter uses a combination of phoneme and word confusion networks.

[HTBRT06] presents a pivot algorithm for creating a normalized generic word confusion network and use it for natural language understanding, named entities and topic classification. In [KINS+12], methods for the extraction of named entities are presented, with a focus on conversational telephone speech by using maximum entropy clustering on the elements of the confusion network.

A system dealing with expressions like “double T” and “M as in Mike” is presented in [ScRK00]. In this publication, the spelling phrases are used for an automatic directory assistance system for a whole country. The system allows selecting cities, names and streets via speech, and — if necessary for clarification — via spelling. This is conducted in a two-stage process: First, a confusion network is parsed using a stochastic context-free grammar consisting of rules for common spellings and spelling alphabets; this way, a bigger graph is created containing the additional spelled information. Then this graph is searched for letter sequences that form the spelled word. A limitation to this word is applied by checking it against a database, e. g. one containing last names.
This is an approach similar to the one used in this work, but has some differences: our system is not limited to recognizing entries from a database; instead of a two-stage approach using a grammar to extend a confusion network, we directly extract the information from the network; furthermore, our system contains more varieties of spelling, supports capitalization, multiple words in one utterance and some more details that allow more human-like spelling.

Another similar system is presented in [BGWT04], where a hybrid two-step process combining statistical tagging with hand-written grammars is used for named entity extraction from word lattices. Statistical taggers have a high recall, whereas grammars have a high precision, thus making it an advantageous combination. This is evaluated in a dialog manager. Furthermore, the paper presents a method for combining knowledge-based (hand-written) grammars with data-induced ones, which might be used for future work on our problem. Thereby, the grammar developed in Chapter 3 might be extended by information from collected data.
It is similar to our work in respect of the specific two-step approach: First, we use a decoder to produce a confusion network (high recall), then we extract exactly matching phrases from it (high precision), as explained in Chapter 4. However, our goal is spelling and not general named entity extraction.

1.4.4 Further Topics

This section discusses some assorted topics related to our work.

For the task of spelling using codewords like “Y as in yankee”, we used existing field-tested spelling alphabets; see Appendix A.3 for a list of them. [LeCo03] presents several user-centered design studies conducted to develop a specialized spelling alphabet for mobile devices. One of the procedures there to find word candidates for spelling a letter is a web-based survey asking people about words that they associate with a letter, which is similar to our approach in Section 5.2. In Section 6.3.1, we also evaluate commonly used codewords from our recording sessions.

Spelling has many use cases, which are presented in Section 3.1.1; an important one for the general improvement of the robustness and quality of speech recognition is error correction. Both [Suhm97] and [SuMW99] compare interactive error correction methods like typing and spelling in terms of accuracy and speed, in order to find out in which situation which method works best. Furthermore, [Moor04] evaluates the speed of different input methods for error correction in ASR, with the peculiarity of comparing single letter spelling and spelling using the NATO alphabet. The NATO alphabet is also used by [FiSe04] in order to recover from errors in a flight reservation

system, where the decoded sequences are checked against a database of e. g. flight destinations.

A speech repair approach using confusion networks is tried out in [OgGo05], where word alternatives are presented to the user on screen so that the correct one can be chosen. This idea is similar to the one used for our live feedback system in Section 4.6: If the decoder is not sure about a letter, alternatives are presented to the user on screen, who subsequently spells the correct one again as a means of error correction.

Another application of spelling is the integration of out-of-vocabulary (OOV) words. In [ChSW03], the acquisition of OOV words is conducted via a multi-stage recognition procedure that extracts pronunciation and spelling (phoneme and grapheme form) of a word that is spoken and spelled consecutively in a single utterance. The word is then added to the dictionary, so that it can be recognized in the next utterance.

In order to exploit this redundancy of speaking and spelling, [MeHi97] present a method for the combination of spoken and spelled proper names. A similar approach is presented in [BaJu99] for recognizing e. g. city names by first trying to recognize the name using a standard word recognizer and falling back to a spelling recognizer if this fails.

This task also relates to the detection of spelled sequences in an utterance that otherwise consists of continuous speech. [HiWa95] engages in localizing spelled letter segments in continuous speech in order to reclassify them using a specialized letter recognizer.

2. Foundations

This chapter provides an introduction to topics that are elementary for this work. First, some basics about linguistics and spelling are presented, then the principles of automatic speech recognition are described with a focus on those components of a recognizer that are relevant to this work. Lastly, the used speech recognition toolkit Janus is introduced.

2.1 Linguistics and Spelling

Linguistics is the science and study of language, where a language describes the interaction between sound and meaning of words. The objective of this work can thus be interpreted as the inference of the meaning of a given spoken utterance in form of the spelled information in it. For this, this section introduces the concepts of letters and alphabets as well as spelling.

2.1.1 Letters and Alphabets

Writing systems are used to both store and communicate ideas. They consist of a set of distinct symbols which encode information when concatenated. There are different categories of writing systems that differ in the way their symbols are used [DaBr96], as shown in Table 2.1.

Type	**Elements**	**Example**
Logosyllabic	morphemes	Chinese characters
Syllabic	syllables or morae	Japanese kana
Featural	phonetic features	Korean hangul
Abjad	phonemes (consonants)	Arabic alphabet
Abugida	phonemes (consonants and vowels)	Indian Devanāgarī
Alphabetic	phonemes (consonants or vowels)	Latin alphabet

Table 2.1: Classification of writing systems

An **alphabet** is a type of writing system whose symbols are called letters. In this work, we use the English alphabet, which is basically identical to the Latin alphabet, and is listed in Appendix A.2 with the pronunciation of its letters.

A **letter** is a written character, or grapheme, which generally represents the sounds of the spoken language the alphabet belongs to. These sounds are composed of **phonemes**, which are a cognitive abstraction representing the smallest structural unit that distinguishes meanings of words. The English language has about 50 distinct phonemes; Appendix A.1 lists the ones used in this work[1].

There are two forms of each letter in the English alphabet: an upper case form, which is also called captial or majuscule, and a lower case form, also called minuscule. They represent the same sound, but serve different purposes in writing: In English, capital letters mark the beginning of a sentence and the first letter of proper names. The implications of the latter for a spelling recognizer are discussed in Section 3.2.

The 26 letters of the English alphabet are divided into vowels and consonants depending on their particular sound. A **vowel** is a sound which is pronounced with an open vocal tract, like "oh", which means that air pressure does not build up. In English, usually the letters 'A', 'E', 'I', 'O', and 'U' are considered vowels. In contrast, a **consonant** is a sound with a complete or partial closure in the vocal tract, like "sh", where the flow of air is constricted. All other letters fall in this category.

In this work, we distinguish between the terms **letter** and **letter name**: a letter denotes the written form of a symbol of the alphabet, like 'Z'; the corresponding letter name represents its pronunciation, like *zee* or /Z IH/.

In the English language, many letters have a similar pronunciation [MaSc94, LeCo03], which means there is a high acoustic confusability between certain letter names. A prominent subset of of the English alphabet that has this characteristic is the so-called **E-set** [MiSe99]. It consists of the letters 'B', 'C', 'D', 'E', 'G', 'P', 'T', 'V' and 'Z'[2], which all contain the *ee* sound at the end and only differ in the short, mostly plosive, consonant in the beginning (except for 'E' itself). They make up more than a third of the alphabet and they are hard to distinguish acoustically, which is known to cause problems in spelling recognition. This is also mentioned in Section 6.3.2 and can be seen in the letter name confusion matrix in Appendix A.4.

Furthermore, more than half of the letters are pronounced like English words (see Appendix A.2 for a list of them), and some more sound very similar to words, like 'S' – "as" and 'N' – "in". This was a particular challenge in this work, as is explained in Section 3.1.2.

2.1.2 Spelling

The term **spelling** has two related meanings: it describes the accepted order of letters in a word, which is also called orthography, and it relates to the act of forming a word by writing or stating its letters [Oxf].

[1] We use a computer-friendly notation instead of the widely-used International Phonetic Alphabet (IPA), which needs Unicode encoding when used in a computer. More information on the IPA at http://www.langsci.ucl.ac.uk/ipa/ (accessed 2014-03-07).

[2] The letter 'Z' is pronounced *zee* in American English only, in most other dialects of English the pronunciation *zed* is used.

In this work, the latter meaning is more important, as our goal is to recognize a sequence of letters. This sequence can however be arbitrary (e. g. spelling a passport number), thus correct orthography is not a primary issue. In this interpretation, the intention of spelling is to communicate an exact sequence of letters from a sender to a recipient.

There are different ways people spell a word or letter sequence in speech. The basic manner is to state the letters themselves like "H E L L O"; in order to add robustness and clarity to this, the acoustic evidence for each letter can be enhanced by adding distinct words to it like "H as in hotel". This may be necessary under hindered communication conditions like telephony or in the presence of ambient noise like road traffic. Depending on the intended recipient, there is a high variety of ways to achieve this, the sender might also say "H is for hotel" or simply "hotel".

A word like "hotel", that is used as replacement or acoustic expansion of a letter, is called **codeword** in this work. A whole statement elaborating on a letter like "H as in hotel" is called a **phrase**. Phrases can also be used to state capitalization ("upper case H") and other purposes, as described in Section 3.2. Thus, we use the term 'phrase' to denote a group of words that has a semantical meaning attached to it[3].

As the reason for the use of codewords for a letter is to aid the recipient in understanding the letter, it is desirable to find words that are acoustically distinct and easy to distinguish [LeCo03]. These are collected in lists called **spelling alphabets**, which provide a codeword for each letter of an alphabet. There is a variety of spelling alphabets; some of them developed naturally, others were designed using empirical or theoretical criteria for the selection of appropriate words. Appendix A.3 lists examples and provides sources for modern and historic English spelling alphabets.

The most widely known and used spelling alphabet is the **International Radiotelephony Spelling Alphabet** [ICA01], also known as the **NATO phonetic alphabet**[4]. It was developed by the International Civil Aviation Organization (ICAO) in the 1950s for international use in radio communication, later adopted by the North Atlantic Treaty Organization (NATO), and today is used by many other organizations as well. The first three codewords are "Alpha, Bravo, Charlie", the full alphabet is listed in Appendix A.3.

Apart from being orthogonal in the acoustic space, the words of the NATO alphabet were designed to be pronounced the same by speakers of different languages. Thus, there are some variations in spelling, e. g. "Juliet" is sometimes written as "Juliett" in order to avoid the silent 't' at the end when spoken by a French speaker. Furthermore, the pronunciation of some words differs from standard English, e. g. "Papa" becomes "Pa-PAH" with the stress on the second syllable and "Quebec" is pronounced "keh-beck" as in French. As most participants in our data collection (see Chapter 5) did not know these rules, some of the words were mispronounced in our test data. In order to avoid these issues, it is also possible to use any other word

[3] This is opposed to the linguistic sense, which uses the term for a group of words with a specific function in the syntax of a sentence.

[4] It is not a phonetic alphabet like the International Phonetic Alphabet (IPA, see Footnote 1); the naming originated historically to distinguish it from e. g. a Morse alphabet, because it is used to spell out messages by voice.

as code for spelling a letter, like “A as in apple”, as long as the recipient is able to distinguish them.

2.2 Automatic Speech Recognition

Automatic speech recognition (ASR) engages in the automatic conversion of spoken human language into the corresponding machine processable text form.

A typical ASR system consists of the components as shown in Figure 2.1. The basic process is divided into the two steps of **preprocessing** and **decoding**.

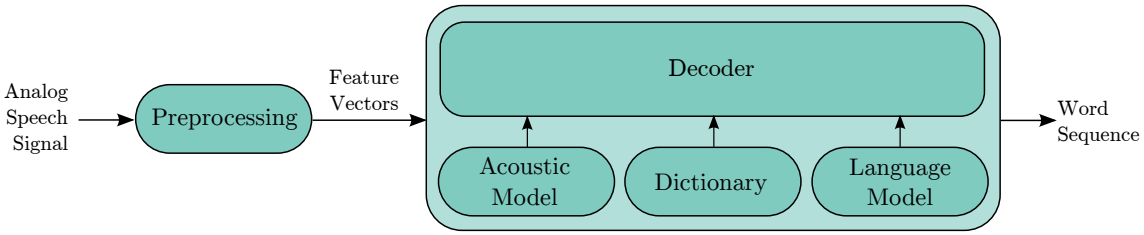

Figure 2.1: Block diagram of a typical speech recognition system

Given an utterance of human speech, a microphone is used to record the sound wave. This signal is sampled to create a digital representation, which the preprocessing uses to produce a sequence of feature vectors describing the most important characteristics of the signal. This is achieved via framing, filtering and transformation into the frequency domain. The objective of this step is to reduce the input signal to its significant features as a preparation for the following decoding process. Each phoneme uttered in the speech signal hence becomes a short sequence of feature vectors.

Provided with the sequence X of feature vectors that represent the utterance, the decoding then finds the most probable word sequence $\hat{W}$, called a **hypothesis**, using the probability measure $\mathrm{P}(W|X)$. Using the Bayes rule, this can be decomposed into the **fundamental equation of speech recognition**:

$$\mathrm{P}(W|X) = \frac{\mathrm{P}(X|W) \cdot \mathrm{P}(W)}{\mathrm{P}(X)} \tag{2.1}$$

In this equation, $\mathrm{P}(X|W)$ is the probability that, given the sequence W of words, the sequence X of feature vectors is observed, which is called the **acoustic model**. $\mathrm{P}(W)$ is the prior probability of observing W, independently from other probabilities, and is called the **language model**. Finally, $\mathrm{P}(X)$ is the prior probability of observing the sequence X of feature vectors.

The decoder then tries to find

$$\begin{aligned} \hat{W} &= \underset{W}{\operatorname{argmax}}\ \mathrm{P}(W|X) \\ &= \underset{W}{\operatorname{argmax}}\ \frac{\mathrm{P}(X|W) \cdot \mathrm{P}(W)}{\mathrm{P}(X)} \\ &= \underset{W}{\operatorname{argmax}}\ \mathrm{P}(X|W) \cdot \mathrm{P}(W) \end{aligned} \tag{2.2}$$

This is done by searching in the space of phoneme and word sequences. The search is commonly limited by the **dictionary**, which defines a set of words and their pronunciation to be used to compose W.

In state-of-the-art systems, the three logical components acoustic model, dictionary and language model are invoked simultaneously, however it is best to describe their functioning separately. The following sections introduce these components, followed by a description of the evaluation process of ASR.

2.2.1 Acoustic Modeling

The objective of the **acoustic model** (AM) is to find the sequence of phonemes that most likely account for the feature vectors of a given audio signal.

Speech can be approximated as a piecewise stationary process in a short time scale of about 15 ms. Therefore, **Hidden Markov Models** (HMMs) [RaJu86] are a common way to describe the acoustic features and their change over time by modeling them as independent states with a stochastic transition process between the states. Typically, each state represents a whole or a part of a phoneme. There are some advantages of using this stochastic model: it can be automatically trained in order to optimally represent given acoustic data, and efficient decoding is possible by using dynamic programming approaches.

Within the HMM states, most commonly **Gaussian Mixture Models** (GMMs) are used to approximate the distribution of a training set of feature vectors derived from audio data. These vectors are occurrences of the particular phoneme (or part of it) that is represented by the HMM state. A GMM is a weighted sum of Gaussian distributions, where a **Gaussian distribution** $\mathcal{N}$ with dimension d, mean vector μ and covariance matrix Σ is defined as in Equation 2.3.

$$\mathcal{N}(x|\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)^d \ \det(\Sigma)}} \cdot \exp\left(-\frac{1}{2}(x-\mu)^{\mathrm{T}} \ \Sigma^{-1} \ (x-\mu)\right) \tag{2.3}$$

Gaussian mixtures allow for a low number of parameters compared to the microstructure of a set of single feature vectors, which makes them robust and easier to use. There are different ways of tying the GMMs to the HMM states; most importantly these are the semi-continuous model, where several states share a set of Gaussians and differ only in their weights in the mixture, and the fully-continuous model, where each state has its own set of Gaussians and weights.

2.2.2 Dictionaries

The **dictionary** is the lexical component that binds the acoustic model and the language model together by providing a mapping between the orthographic form of a word and its phonetic representation.

The acoustic model deals with sequences of phonemes. In order to convert these into proper words, the dictionary stores both forms of each recognizable word. This also means that words are unrecognizable if they are not in the dictionary. On the one hand, this helps the decoder to narrow the search space down; on the other hand, it causes recognition errors for words that are **out of vocabulary** (OOV). These are some exemplary entries:

```
boat              {{B WB} OW {T WB}}
faster            {{F WB} AE S T {AXR WB}}
penguin           {{P WB} EH NG G W AX {N WB}}
```

The phonemes that are parenthesized together with the word break tag `WB` mark the beginning and the end of each word. For a list of all phonemes used in our systems, see Appendix A.1.

2.2.3 Language Modeling

The **language model** (LM) is a component that tries to ensure the right choice and order of words in a sentence. For this, it examines a sequence of words or the whole sentence, assigns a probability to it and thus can exclude improbable or wrong hypotheses. Its goal is to produce a "good" sentence, i. e. fluent language and correct syntax, thus it has a notion of grammaticality [Bell04].

A language model has the ability to distinguish between words that are pronounced similar but spelled differently, e. g. "seas" and "seize", which are called homophones. It usually does so by embracing the surrounding words and deciding for the spelling that is more probable in the given context. As many letters of the English alphabet are also words or sound similar to words, this is particularly important to distinguish e. g. the phrase "A as in apple" from "A S N apple".

Two typical types of language models are the n-gram LM, which uses a probabilistic approach, and the Context Free Grammar LM, which is based on formal languages. Both are presented in the following paragraphs.

n-gram Language Models

The ***n*-gram Language Model** assigns a probability to a given sentence by splitting it into sequences of n words whose probabilities are easier to handle than a whole sentence.

The objective of an LM is to aid the decoder by assigning a probability to each produced hypothesis. As the number of possible sentences in a natural language is too high to be trainable and manageable, the LM has to be broken down into smaller chunks. A chunk of n words is called an n-gram; it can be seen as a word in the context of its $n-1$ preceding words. For example, the sentence

> The fox eats a salad.

is split into the 3-grams (also called trigrams)

- <s> <s> the
- <s> the fox
- the fox eats
- fox eats a
- eats a salad

- a salad </s>
- salad </s> </s>

The symbols <s> and </s> are fillers for the beginning and end of a sentence in order to have proper trigrams there. The probability of the whole sentence is then calculated as the product of the trigram probabilities.

In general, given a sentence of m words $w_1, \ldots, w_m$, the probability of observing this sentence using an n-gram model is approximated according to Equation 2.4.

$$\mathrm{P}(w_1, \ldots, w_m) = \prod_{i=1}^{m} \mathrm{P}(w_i | w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} \mathrm{P}(w_i | w_{i-(n-1)}, \ldots, w_{i-1}) \tag{2.4}$$

The right-hand side of the equation is a grouping of words with the same history of preceding words (the same context) into a class with a shorter history of $n-1$ words.

The conditional probability of the n-grams can be estimated from a given text corpus by counting the n-gram frequencies as shown in Equation 2.5.

$$\mathrm{P}(w_i | w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\mathrm{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\mathrm{count}(w_{i-(n-1)}, \ldots, w_{i-1})} \tag{2.5}$$

This estimator is used for training the LM on a text corpus by counting combinations of words and estimating the probabilities of n-grams from that. The higher n is, the more text is needed in order to achieve a reliable and robust estimation, because longer contexts are more infrequent.

Context-Free Grammars

A **formal language** is a set of strings built from a finite set of symbols, called the alphabet [Chom56, Chom57]. A string that consists solely of symbols from that alphabet and that is part of the language is called a valid or well-formed sentence of the language.

In order to define a formal language, i. e. , the set of valid strings, often a **formal grammar** is used which provides formation rules for the strings. A grammar G is defined by the 4-tuple $G = (V, \Sigma, P, S)$, which consists of the following components:

1. V is a finite set of nonterminal symbols that represent types of clauses in the sentences build from G.
2. Σ is a finite set of terminal symbols that is disjoint from V. These symbols build the actual content of the sentences and thus are the alphabet of the language defined by G.
3. P is a finite set of production rules of the form

$$(V \cup \Sigma)^* \; V \; (V \cup \Sigma)^* \rightarrow (V \cup \Sigma)^*$$

where the asterisk represents the Kleene star operation, meaning the expression left of the operator can occur arbitrarily often, also zero times. Each production rule maps from a string of symbols to another, where the left-hand side must contain at least one nonterminal symbol.

4. $S \in V$ is a distinguished start symbol from the set of nonterminals that is used as the initial point for building a sentence.

The production rules are used to replace nonterminals by terminals. First, the start symbol, being a nonterminal, is replaced according to a rule where it appears on the left-hand side. Then, further production rules are applied, until there are no nonterminals in the sequence left. This process is used to generate sequences of terminal symbols. The choice of rules can either be random or following a certain scheme, e. g. , sequential, so that all possible sentences of the language can be built (see the paragraph on probabilistic context-free grammars for a variation of this).

Vice versa, given a sequence of terminals, a grammar can also be used to prove that the sequence belongs to a formal language. This process is called derivation and results in a parse, which is a sequence of production rule applications that transform the start symbol into the given sequence of terminals.

There are different levels of complexity for these languages, called the Chomsky hierarchy, that differ in the allowed production rules. Depending on the level, different types of grammars are needed to describe the language.

The type of grammar that is most important in this work is called **context-free grammar**. The name expresses that the formation rules of this grammar do not take context into account, which means that the set P consists of rules of the form

$$V \to (V \cup \Sigma)^*$$

where the production rules of the nonterminals can be applied regardless of the context of them.

In speech recognition, a context-free grammar can be used as a language model that makes hard decisions about which sentences are valid and which are not. It thus restricts the utterances a user may say, and is usually used for a limited vocabulary and in special applications. On the one hand, it is manual work to develop the formal language by writing the rules; on the other hand, it then is easy to attach some semantic meaning to the rules, which is useful in dialog systems and other applications where a downstream analysis of the utterance is needed, as discussed in Chapter 3.

JSpeech Grammar Format

In this work, mainly the **JSpeech Grammar Format** (JSGF) is used as a notation for grammars and their rules[5]. It is specialized for purposes of speech recognition, which also means that the used symbols are whole words.

[5] The full specification is available at http://www.w3.org/TR/jsgf (accessed 2014-01-30).

The sets of symbols V and Σ are not given explicitly in JSGF, but derived from the words being used in the production rules. Nonterminals are denoted in angle brackets like `<nonterminal>`, while all other words are terminals. To separate alternatives, the syntax `...|...` can be used; square brackets like `[ ... ]` enclose a sequence of optional symbols; parentheses like `( ... )` are used for grouping of symbols. An exemplary excerpt from a grammar in JSGF looks like this:

```
<sentence>  =  <subject> <verb> <object>;
<subject>   =  the <animal> | the rock | the man [ who likes bacon ];
<animal>    =  fox | elephant | [ black | white ] cat;
<verb>      =  eats | talks to | sees;
<object>    =  the @fruit | the ( red | green | blue ) house;
```

Here, the terminal `@fruit` is starting with the `@`-tag, which is an extension to the standard JSGF that marks word classes. A terminal with the word class tag can be replaced by a corresponding list of words, so from a logical point of view it is a nonterminal. However, using this notation allows for shorter grammars and reading convenience.

The JSGF notation is used for our grammar-based spelling recognizer in Section 3.3.2.

Probabilistic Context-Free Grammars

In general, when applying the production rules of a grammar, it is ambiguous which rule is selected given a sequence of terminals and nonterminals. Thus, it is a non-deterministic process that may lead to different outcomes depending on the order of rule applications.

In order to influence the process, a grammar can be extended by assigning probabilities to the production rules. The sum of the probabilities of all rules with the same left-hand side has to be 1. Each sequence of rule applications — like a derivation of a terminal sequence — then gets a probability, which is the product of the probabilities of the applied rules. This probabilistic component helps dissolving ambiguities.

Furthermore, a probabilistic grammar can be used to generate terminal sequences where specific phrases — modeled as production rules — occur following an arbitrary distribution. In Section 4.3, a **Probabilistic Context-Free Grammar** (PCFG) is used to generate sentences of spelled expressions in order to train an LM on it. PCFGs extend context-free grammars in a similar way how HMMs extend regular grammars.

2.2.4 Lattices and Confusion Networks

Modern ASR systems are based on stochastic models using multiple decision criteria. This allows to avoid hard decisions during the decoding in order to take into account all sources of knowledge before finally settling at one hypothesis.

A common representation of the alternative candidates that the decoder is considering in the decision process is the **word graph** or **lattice** [OeNe93, KHBCB+07b]. It is a directed acyclic weighted graph where the nodes belong to points in time

of the utterance, and the edges are labeled with a word. The weight of the edges (often called **score**) comes from the posterior probabilities of the decoder. A node is dedicated to each the start ⊙ and end ⊚ of the utterance, so that each path between those two represents a hypothesis. Figure 2.2 shows an exemplary lattice.

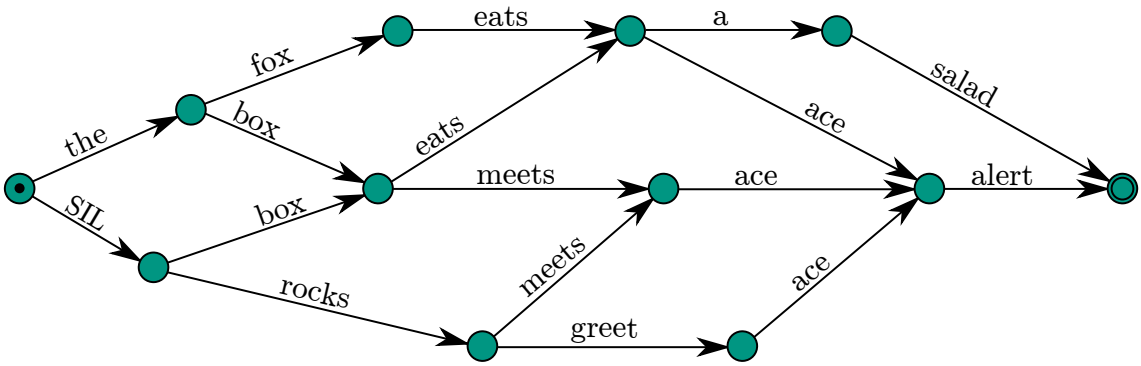

Figure 2.2: A lattice for the sentence "the fox eats a salad"

Any finite sequence of words can be encoded in a lattice; in general, an exponential number of sentences can be stored in polynomial space. By scoring the paths through the lattice, an n-best list of sentences can be extracted, where the best one usually is outputted as the final hypothesis of the decoder. However, also the lattice can directly be used as output, which then forms an efficient interface between speech recognition and downstream analyses like linguistic processors.

As a lattice usually has a complex topology, for many applications an approximation is used instead, which is called a **confusion network** [MaBS00, KHBCB+07a]. The two have their structure of a directed acyclic weighted graph with distinguished start and end nodes in common. A confusion network however has the additional peculiarity that each path between the start node ⊙ and end node ⊚ goes through all nodes of the graph. Figure 2.3 shows such a network.

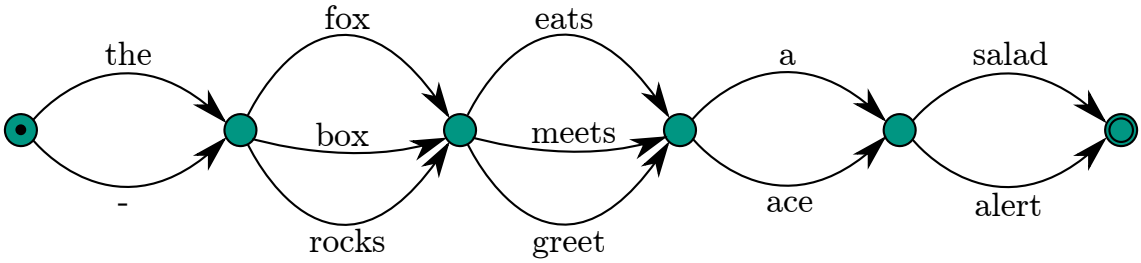

Figure 2.3: A confusion network for the sentence "the fox eats a salad"

Confusion networks contain all paths of the original lattice and may contain further paths, which allows for new word sequences. Using the special empty word "–", an edge can be skipped, so that it produces no output and thus paths of different lengths are possible.

The edges again have a weight or score which is inferred from the posterior probability of the decoder, and also can be seen as a confidence of the decoder in the word. Between two consecutive nodes, the weights are often normalized, so that they sum up to 1. A path from the start node to the end node is scored as the product of the

scores of its edges, which represents the likelihood of the path. The sum of all path scores is then 1.

The strict structure of a confusion network suggests to interpret it as a sequence of word lists, where each list contains alternatives of similar words that correspond to a time period in the utterance. These lists of words that are separated by nodes are called **segments** of the confusion network in this work. Figure 2.4 shows an example.

$$\left\{\begin{array}{c}\text{the}\\ \text{-}\end{array}\right\}\left\{\begin{array}{c}\text{fox}\\ \text{box}\\ \text{rocks}\end{array}\right\}\left\{\begin{array}{c}\text{eats}\\ \text{meets}\\ \text{greet}\end{array}\right\}\left\{\begin{array}{c}\text{a}\\ \text{ace}\end{array}\right\}\left\{\begin{array}{c}\text{salad}\\ \text{alert}\end{array}\right\}$$

Figure 2.4: A confusion network with segments presented as word lists

This representation is easy to read and stresses the aspect of confusable words, whereas the graph representation stresses the aspect of paths between nodes. Confusion networks are used as the starting point of our main system described in Chapter 4.

2.2.5 Evaluation

The quality of a speech recognition system is often measured in the dimensions of speed and accuracy.

The speed is measured in form of a **Real Time Factor** (RTF) that describes how long it takes to decode an utterance compared to its length. This is important in live systems, where real time decoding is required, which is equivalent to a RTF ≤ 1.

For a given utterance, the desired output of a speech recognizer is called the **reference**, which often is a manual transcription of the words said in the utterance. The reference provides a means of measuring the accuracy of a decoder by comparing it to the hypothesis. A metric for measuring this is a target function, whose optimization is an important goal when improving speech recognition systems.

The accuracy is frequently quantified in form of the **Word Error Rate** (WER), which is a modification of the **Levenshtein distance** [Leve66], also called the edit distance. Given a reference with n words, let s denote the number of substitutions, d the number of deletions and i the number of insertions of words that are minimally needed to transform the hypothesis into the reference. Then the word error rate is calculated as in Equation 2.6.

$$\text{WER} = \frac{s + d + i}{n} \tag{2.6}$$

The WER is helpful to evaluate the immediate word sequence of an utterance. In case of spelling however, it is more significant whether the parsed result of the spelling recognizer is correct; for example, given the utterance “F as in friend, O, X as in X-ray”, the decoded letter sequence “fox” is the one that needs to be accurate. Thus, a method of evaluation on a per-character level is needed.

The Levenshtein distance already is such a metric. In order to illustrate the connection to the default WER, in this work it is called the **Character Error Rate** (CER). It can be calculated efficiently via dynamic programming in form of a matrix recurrence. Given two strings u and v (in our case, these are the reference and the hypothesis), the distance matrix D is defined as in Equations 2.7 to 2.10.

$$D_{0,0} = 0 \tag{2.7}$$

$$D_{i,0} = i, 1 \leq i \leq |u| \tag{2.8}$$

$$D_{0,j} = j, 1 \leq j \leq |v| \tag{2.9}$$

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & +0 \quad \text{if } u_i = v_j \\ D_{i-1,j-1} & +1 \quad \text{(substitution)} \\ D_{i,j-1} & +1 \quad \text{(insertion)} \\ D_{i-1,j} & +1 \quad \text{(deletion)} \end{cases}, 1 \leq i \leq |u|, 1 \leq j \leq |v| \tag{2.10}$$

By tracing the path of the selected minimal cells back through the matrix, it is possible to detect which letters are left out, which ones are inserted and which ones are replaced by others. This is a useful source of information to find out more about particular groups of letters that are easy to confuse, like the E-set. Sometimes, multiple paths lead through the matrix, which means there are different ways to edit the one string to get the other.

Figure 2.5 shows an example of the Levenshtein distance matrix for the strings “dancing” and “distance”.

	ϵ	**d**	**i**	**s**	**t**	**a**	**n**	**c**	**e**
ϵ	(0)	1	2	3	4	5	6	7	8
d	1	(0)	(1)	(2)	(3)	4	5	6	7
a	2	1	(1)	2	3	(3)	4	5	6
n	3	2	2	(2)	3	4	(3)	4	5
c	4	3	3	3	(3)	4	4	(3)	4
i	5	4	3	4	4	(4)	5	(4)	(4)
n	6	5	4	4	5	5	(4)	(5)	(5)
g	7	6	5	5	5	6	5	(5)	(6)

Figure 2.5: An example of the Levenshtein distance matrix

The number in the bottom right corner represents the distance between the two strings. The possible back trace paths are highlighted; one of them leads to the following alignment of the strings

```
d---ancing
distanc-e-
```

This alignment shows how the single letters are edited to transform the strings. It can also be used to evaluate detailed statistics on single letters in form of frequent confusions, insertions and deletions, which is discussed in Section 6.3.2.

2.3 Janus

The **Janus Recognition Toolkit** (JRTk)[6], in this work referred to as **Janus**, is the general-purpose speech recognition toolkit [FGHK+97, LWLF+97] that was used as a basis for the systems built in this work. It is developed by the Interactive Systems Laboratories at Carnegie Mellon University (USA) and Karlsruhe Institute of Technology (Germany); we used version 5.1.1 and additionally expanded it by integrating new code for the parsing component (see Chapter 4).

For building a state-of-the-art speech recognizer, Janus provides a flexible Tcl/Tk[7] interface with an object-oriented approach: The Tcl scripting language is used to access and manipulate objects like HMMs and n-gram LMs and thereby enables developers to quickly build a running system using the given methods of speech recognition in Janus.

A lot of functionality is already present in Janus. Using the Tcl interface, we implemented many algorithms and processes of this work. Janus also provides a one-pass decoder component [SMFW01] which can be configured and accessed via Tcl and was used for all our decodings, both with n-gram language models and context-free grammar language models.

In order to implement the additional functionality needed for this work, particularly the parsing component of Chapter 4, modifying the C source code of Janus was necessary. Due to requirements in speed, the interpretation of Tcl scripts would have been too slow; a convenient secondary effect of this is the possibility to access our parser like any other object in Janus.

[6] See http://isl.anthropomatik.kit.edu/english/1406.php for more information on Janus and its licensing (accessed 2014-02-10).

[7] More information available at http://www.tcl.tk, the source code at http://tcl.sourceforge.net (both accessed 2014-02-26).

3. Preliminary System Development

This chapter describes the preliminary development that was necessary in order to build the main system. First, an analysis of the aspects of spelling and its challenges is given. Then, a section on the spelling language presents the phrases the we modeled for spelling and that are incorporated in our systems, before two baseline systems are described that we built for evaluation purposes. The description of the implementation of our main system is then subject of the next chapter.

3.1 Analysis

This section discusses some aspects and use cases of spelling in speech recognition and introduces the problem background of this task.

3.1.1 Spelling in Speech Recognition

Historically, in speech recognition the decoding of single letters and sequences of single letters was one of the early tasks to be worked on. Section 1.4.1 presents a short review of different approaches to it in the course of time. Given enough training data, modern ASR systems now provide continuous speech recognition with high accuracy. This makes it possible to extend the recognition of spelled letters to a more natural way of spelling as people use with one another when communicating complex words or random letter sequences, for example by using codewords.

There are advantages of this to both the human and the computer side: the user is more accustomed to using this natural way of spelling, for example when spelling ones name; the computer has more acoustic evidence per letter and can apply some other methods to improve the recognition accuracy. Such a system has a variety of applications and use cases:

Error correction: Although there is constant progress in speech recognition, it is not perfect yet — and maybe never will be, because misunderstandings even

happen to humans. Thus, a way of correcting errors and recovering from them is necessary, which can be done via spelling. This may be necessary in an environment where keyboards are not an option and the recovery has to be achieved by voice only, for example in a car. Section 1.4.4 presents some previous work on this topic.

OOV handling: As a language evolves, new words find their way into it. An ASR system thus will encounter unknown words (Out Of Vocabulary) from time to time; spelling provides a convenient way to introduce them to the system. Section 1.4.4 also covers approaches to this, particularly the work presented in [ChSW03].

Dictation: Professional dictation software provides a fast and reliable way of inputting large amounts of text, particularly when the system is adapted to the user and the domain is limited, like in law offices or medical practice. However, sometimes complex inputs require spelling, for example passwords, foreign words, proper names or technical terms[8], which is an application related to OOV handling.

Voice controlled systems: There is a wide range of systems that use speech as input for controlling actions.

- In voice-directed warehousing, the consignment of goods is conducted using pick-to-voice input from the workers. For example, serial numbers of products need to be spelled here. A further advantage of this scenario is the possibility to train worker to the system in order to improve its accuracy, which is also discussed in Section 6.1.2.
- In dialog systems like flight or conference booking, journey planners and other similar situations, spelling is a convenient way to communicate flight numbers, passport numbers, airport codes, destinations, license plate numbers and so on.
- More and more home appliances and customer devices use speech as input. Smart TVs and home entertainment systems can be controlled this way for example. In smartphones, calls can be made using speech to determine the recipient. In these cases, spelling might be necessary for entering new names in the contact list or a new phone number.

These scenarios can make use of spelling as a way of inputting special words or random letter sequences. A subtask needed for the practicability of many live applications is to find these spelled sequences in an utterance that otherwise contains continuous speech, which is for example discussed in [HiWa95], also see Section 1.4.4. However, in this work we focus on the recognition of the spelling sequence itself.

This task combines aspects of speech recognition and natural language processing, as we use the output of an ASR system to extract phrases of spelled information. A major challenge is the high variety of expressions that are used to spell; some examples are shown in Table 3.1.

[8] Speech input is even used to write programming code, as for example demonstrated in http://www.youtube.com/watch?v=8SkdfdXWYaI (accessed 2014-03-13), where short random phoneme sequences similar to letter names are used as commands for the VI editor.

Expression type	Example
Single letters only	"A B C. . . "
NATO Alphabet	"delta oscar golf"
NATO Alphabet phrases	"E as in echo, B as in bravo"
All words phrases	"E as in elephant, B as in boat"
Capitalization	"lower case K", "capital F"
Proper names	"my name is Frey, F as in fox. . . "
Numbers	"seven", "the number nine"
Multiple words	"next word", "space"
Special characters	"hash tag", "exclamation mark"
Diacritics	"E circumflex", "A grave"

Table 3.1: Expression types of spelling

These expressions were inferred from the training data in Section 5.2 and used as basis for the development of the spelling language presented in Section 3.2, which contains all phrases our systems are able to process. We did though not include the latter two expressions — special characters and diacritics — because their use is limited to rare occasions like passwords respectively words in languages other than English, thus they are not essential for this work. However, they can easily be implemented by adding appropriate phrases and extending the algorithm that creates the output, which is described in Section 4.4.2 and Section 4.4.3.

3.1.2 Problem Background

Although single letters and spelling were early tasks in speech recognition, they are still a particular challenge for modern systems. The two main issues are that letter names in English are short, thus provide little acoustic evidence, and that there are many possible confusions, for example of letters of the E-set, as explained in Section 2.1.1. This might for example be a problem when spelling is used in a voice-over-IP system, where packet loss can lead to interrupted speech so that e. g. the plosive sounds at the beginning of E-set letters are dropped. Furthermore, telephones have a cutoff frequency of 8 kHz, so that some sounds get confused: e. g. 'F' and 'S' have their distinctive characteristics mainly in the spectrum above that frequency.

When allowing codewords to build longer phrases for expressing a spelled sequence in a speech recognition system, these problems can be avoided, supposed the user actually utilizes these phrases and does not stick to single letters. There is more evidence per letter and codewords are usually chosen to be orthogonal in the acoustic space to prevent confusions. However, there are other problems in this case.

On the one hand, the system should include many words to be used as codewords for letters, in order to make even improbable expressions like "P as in platypus" possible. On the other hand, if using too many words — like the whole dictionary — without controlling their probabilities with an LM, there may occur errors due to the decoder "finding" short particle words when the utterance actually did not contain them. In the system using a context-free grammar as LM, this occurred, as discussed in Section 3.3.2 and Section 6.1.2.

Another recurring problem in this work is that in English many letters are pronounced the same as words. For example, the confusion of the letter names of 'S' and 'N' with the words "as" and "in" causes that phrases such as "A as in apple" are sometimes interpreted as "A S N apple". This is also discussed in Section 2.2.3; a solution statement is presented in Section 4.5.1.

All these issues are related to single letters and codewords used for letters. Another issue emerges when these letters are composed to the final spelled sequence. In some applications like flight booking, it is possible to check this sequence against a database of possible outcomes, which was for example done in [ScRK00] and [FiSe04], as mentioned in Section 1.4. Other use cases, like the input of proper names, may allow the usage of a large background dictionary of names in a similar manner.

However, we wanted to build a general-purpose spelling recognizer, so basically any arbitrary sequence of letters is allowed without constraints. The learning of OOV words is a good example of a case in which no spell check is possible, precisely because the spelled word is out of vocabulary. Also, the use of a language model is limited: It may help to work out probabilities for single phrases, as described in Section 4.3, but it cannot be used to predict the probability of certain letter combinations in general. For example, passwords are totally random[9].

3.2 Spelling Language

One central aspect of this work is the language that the systems are able to recognize and parse. In order to be computable, it has to have certain restrictions compared to human-to-human spelling; particularly semantic relations and analogies as for example "My name is Night like day" are hard to interpret for a computer system. Thus, a limitation to certain recurring patterns with known semantic meaning is necessary. There are several reasons not to use an automatically inferred formal language in this case, which are explained in Section 1.4.2.

In this work, the patterns to be recognized are called **phrases** in order to emphasize their reference to natural languages. To find out which phrases are necessary for spelling in English, real data was used. In Section 5.2, the collection of training data is described. The 12 native American English speakers delivered more than 1,000 utterances from which the following types of phrases for spelling were inferred.

Single Letters
: This is the simplest and most basic type of phrase, which covers just a single letter like "A" or "B". If such a phrase occurs in an utterance, no further processing is necessary, as the letter can directly be included in the output.

Single Words
: This type expects a single word used as expression for a letter and transforms it into the corresponding letter. In order to prevent false positive matches for all kind of words that appear in an utterance, this type is only used for the NATO alphabet in our systems. This makes it for example possible to spell "fox" using the sentence "foxtrot oscar x-ray".

[9] Hopefully.

"Letter Word"
A type for phrases like "A apple". As only one participant in the training data collection occasionally used this kind of expression and as it created a lot of false positives in other cases, this type was not used in the final version of the systems.

"Letter ... Word"
This is the type of phrase for expressions like "G as in golf" or "M is for milk". Most prominently, it is used with the pattern "as in", but also supports some other variations like "L stands for light" or "P like papa", which however are rarely used.

Word Introduction
This phrase type is used to allow for multiple words being spelled in the same utterance. Hence, whenever it occurs, a space is inserted into the output letter sequence. Possible indicators for this are simply "space" or "blank", as well as "first word" or "the next word is".

Name Introduction
This type is used for indicating proper names. It has two different implications for the output: First, it can be used to separate different parts of a name, similar to the word introduction: By uttering "my first name is ..., middle name ..., the last name is ...", three words separated by spaces are outputted.

The second implication of this expression is the possibility of automatic capitalization: Names usually have capital initial letters followed by lower case letters for the rest, which our systems automatically transact — unless otherwise declared (for example in case of Scottish surnames like "MacFarlane"). In a similar manner, also streets names etc. could be added to our system.

Modifier: Upper Case/Lower Case Letter
This kind of phrase occurs when the speaker wants to indicate that the following letter has some special capitalization. For example, when uttering either "capitalized", "capital" or "upper case", the following letter will be put to upper case. This is helpful when spelling passwords, where each letter may have different capitalization.

Modifier: Upper Case/Lower Case Word
In order to indicate that the whole following word is either upper or lower case, this kind of phrase can be used. A possible indicator for this is "next word is all upper case"; however, because of the rare occasions where this is necessary, this type of phrase was not used in our final systems.

Modifier: Upper Case/Lower Case Sentence
The case of the whole sequence of spelled letters having the same capitalization is more frequent. This phrase type has a lot of possible ways to be used in English, among them: "all of this is in capital letters" or "everything in caps".

Modifier: Double Letter
This modifier causes the following letter to be doubled, so that e. g. "double Q as in quartermaster" becomes "qq". This kind of expression is more common in e. g. German than in English, but nevertheless useful in some cases.

A particular difficult situation arises from the homophony of the phrase "double U" and the letter 'W', being pronounced *double-u*. The situation rarely occurs, but is an unsolved issue of this work.

Numbers
This phrase type stands in for any number used in spelling e. g. passwords and matches either "the number x" or simply "x" itself. As this is a minor issue of this work, our systems currently support integer numerals between 0 and 99 only.

According to our training and test data, these types of phrases suffice for all encountered cases of spelling in the English language. Of course, this depends on the task and kind of strings that have to be spelled. Also, in other languages than English, additional or completely different phrases may be needed, as is the case for e. g. Portuguese, where diacritics are necessary [RoRM97].

The phrases are assembled into a spelling language by simply concatenating them to one another. Parts of an utterance that are not matching any phrase type are interpreted as fillers and are ignored when being parsed.

Depending on the type of the system, different ways of representing the language are used. The system based on context-free grammars in Section 3.3.2 uses a grammar that includes all the phrases, while the main system of this work directly finds phrases in a confusion network, as described in Section 4.4.

3.3 Baseline Systems

Before we started to work on the main system, we used the given capacities of Janus to develop additional systems: As the software infrastructure of Janus is ready to use, it was easy to build prototypes for quickly testing out ideas and getting some basic knowledge about recognizing spelled speech. From these prototypes, we developed two systems that are used as a baseline in the evaluation. They are build using a standard Janus core, so that they can be reimplemented and tested by others as well.

The two baseline systems are presented in the following sections; the first one recognizes single letters with an n-gram LM, the second one uses a context-free grammar as language model to describe a more complex spelling structure.

3.3.1 Single Letters

In the history of speech recognition, single letter decoding was one of the early tasks to be worked on; see Section 1.4.1 for more on this. On the one hand, it is straightforward, because the vocabulary is limited to the 26 letters of the alphabet. On the other hand, due to the lack of context in form of a proper language model and the little acoustic evidence per letter, it may still be a challenge even for state-of-the-art recognizers.

In order to see how a modern ASR system — with all its improvements compared to the beginnings of speech recognition — copes with this task, the first system we

developed for this work was a single letter decoder. Janus is usually a continuous speech recognizer, but in this task it had to decode sequences of spelled letters.

The system structure is as shown in Figure 2.1 in the introduction of automatic speech recognition in Section 2.2, which is the standard way of using Janus. The acoustic models were taken from previous experiments at the Interactive Systems Labs and trained on the Wall Street Journal corpus[10] (WSJ corpus). The dictionary consists of the single letters of the English alphabet and some special characters like "space", "hyphen" and "period" (which are not relevant for our case, as they do not appear in the test data). The language model is a 4-gram model containing the frequencies of single letters and special characters, also trained on the WSJ corpus. It is however questionable whether an n-gram LM for single letters is reasonable, because in many cases of spelling, the sequence of letters is more or less random, so that a statistical LM might not be helpful.

This system is used as a starting point for further development and has the ability of recognizing consecutive single letters like "H E L L O". Thus, it is not able to perform the desired task of decoding human-to-human spelling sequences. However, it is useful as a reference and baseline for the following more complex systems and thus helpful for proper evaluation.

3.3.2 Context-Free Grammar

The task of spelling has a relatively strict structure as described in Section 3.2. A context-free grammar is therefore a sufficient model to realize this structure in form of a formal language (Section 2.2.3 introduces the use of a CFG as a language model).

We thus built a CFG decoder that is structured as in Figure 3.1. The blue parts symbolize the components that are exchanged with regard to the standard decoder.

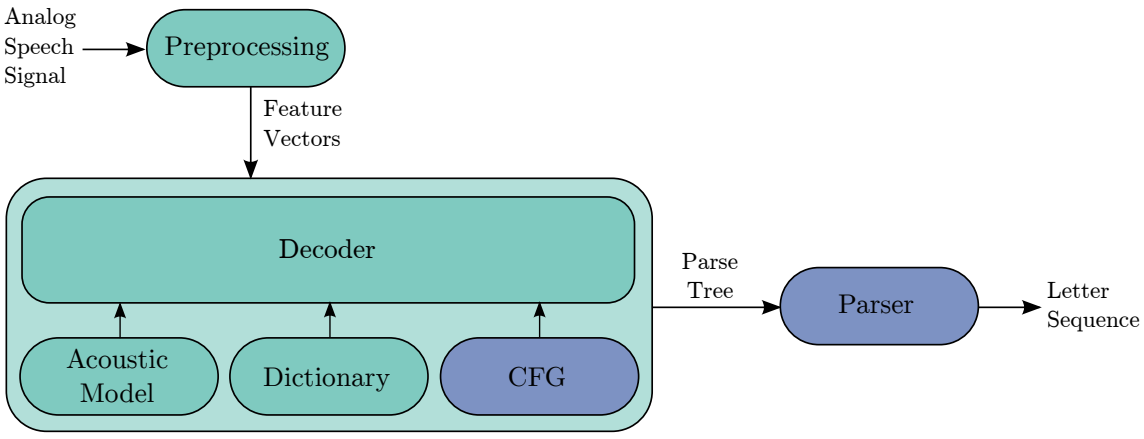

Figure 3.1: Block diagram of the CFG system

A further advantage of the development of such a decoder is that a CFG-based language model is already available in Janus, thus the main work here was to write the grammar for spelling based on the phrases presented in Section 3.2. The full

[10] Available at http://catalog.ldc.upenn.edu/LDC2000T43 (accessed 2014-02-10).

grammar written in JSGF can be found in Appendix B.1; see Section 2.2.3 for an introduction to the JSGF specification. The placeholders for word classes like `@letterword-A` that are found in the grammar are filled in with the words from the lists described in Section 4.3.2; this allows the user to use almost any word as a code to spell a letter.

The acoustic model of the system is the same as used in Section 3.3.1. As the decoder is restricted to the words that appear in the grammar, the dictionary can also be reduced to these words. It is automatically generated by intersecting a large background dictionary with the set of words in the grammar.

The system output is limited to sentences that are part of the language of the grammar. Furthermore, instead of a hypothesis, the decoder now outputs a tree symbolizing the rules and their order that were applied during decoding of the utterance. It is called a **parse tree**, as it shows how the decoder parses the content of an utterance in order to fit into the grammar. Given the full grammar of our systems, a possible parse tree looks as shown in Figure 3.2.

```
<utterance> (
    <word> (
        <char> (
            <number> (
                <number-6> ( six )
            )
        )
        <char> (
            <letter> (
                <letter-O> ( O as in otto)
                <case-modifier-upper> ( capital )
                <letter-H> ( H )
                <letter-K> ( K as in kite )
                <letter-J> ( J )
            )
        )
    )
)
```

Figure 3.2: An example of a parse tree of the CFG system

The innermost parts of the parentheses are the terminal symbols of the grammar, which are the words that were actually uttered, whereas the nonterminals in angle brackets are an abstraction that shows the derivation of the rules.

In order to get the proper spelled sequence `6oHkj` from this, a second step of parsing is necessary that extracts the sequence by traversing the tree. This is the parser component shown in Figure 3.1. The parser is a script adjusted to the grammar that iterates over the parse tree and uses flags for modifiers like `<case-modifier-upper>` to ensure the correct application of the semantics of the phrases. The output is then the spelled sequence of letters.

Having a parse tree with semantic meaning is of advantage in dialog systems, where this can directly be used to interpret user input and infer actions from it. This is for example the case in restricted domains like passport number detection or warehouse systems.

On the one hand, the limitation to sentences of a fixed formal language makes it easy to parse the output of the decoder, as the structure of the sentences is known. On the other hand, it may be too strict a limitation for many real life scenarios. Particularly parts of the utterance with no semantic meaning like hesitations ("ehm, well, yeah,...") and other words not belonging to the actual spelling sequence ("my friend's dog is named...") are a cause of problems: It is hard to add all possibilities and variations that a user might utter. In addition, the decoder tries to find a suitable derivation whenever possible, so that non-matching parts are often squeezed into the grammar, resulting in wrong derivations with low probability, that are causing errors in the result.

Furthermore, if too many such small snippets are modeled, it may come to overfitting, where the decoder wrongly detects words in the utterance, that indeed are in the grammar, but make no sense in context. This effect also occurred because of the large background dictionary we used to fill in for the word classes like `@letterword-A`; see Section 6.1.2 for an example of this effect, which most likely occurs because the grammar has no probabilities for these snippets like an n-gram LM would have.

Another issue with a CFG language model is the unpredictable time that is needed to decode an utterance. The complexity of finding a derivation depends on the amount of word alternatives that the decoder tries out, which in turn depends on many factors like clearness of speech, and thus varies between speakers and utterances. There is no apparent correlation between the length of an utterance and the time that is needed for decoding; some utterances were decoded within seconds, while others needed to be stopped after hours of unsuccessful computation. In the evaluation, those utterances causing a time problem were therefore ignored. This is also discussed in Section 6.1.2.

Considering the drawbacks mentioned above, a more flexible approach is needed, which is the topic of the following chapter.

4. Phrase Network System

This chapter describes the Phrase Network System, which is the name we gave to the specialized system developed as the main part of this work. Its goal is to combine the rich structure of a context-free grammar with the flexibility and sophistication of a state-of-the-art continuous speech decoder.

The chapter first gives an overview to the system, then describes the decoder with some elaboration on the language model. The parser is the component that then extracts the desired information. Afterwards, some improvements are described, as well as a live feedback system for use in real life applications. Finally, a short summary is given for the whole system.

4.1 Overview

The task of recognizing human-like spelling involves both Automatic Speech Recognition and basic Natural Language Processing. Also, for some applications a dialog system build around the recognizer may be an interesting additional feature. Therefore, it is desirable to have a flexible approach that can be adapted to the needs of different situations and still makes use of given linguistic structures as much as possible in order to maximize accuracy.

Using a decoder with a fixed context-free grammar thus is a useful approach when the users can be expected to follow the rules of the grammar, as described in Section 3.3.2. This is for example the case in pick-to-voice systems in warehousing, where workers can be trained to use special vocabulary. The more restricted the language, the lower the error rate that can be reached, as is shown in Section 6.1.2. However, the average user most likely does not know a full spelling alphabet and may want to use a looser way of spelling. Therefore, are more flexible approach is necessary.

The system developed for this work embeds a strict structure into an adjustable framework in order to combine both merits. It is named **Phrase Network System**, as the main idea is to extract the desired information from a network of interconnected phrases derived from the confusion network of a standard decoder. The general structure of the system is shown in Figure 4.1.

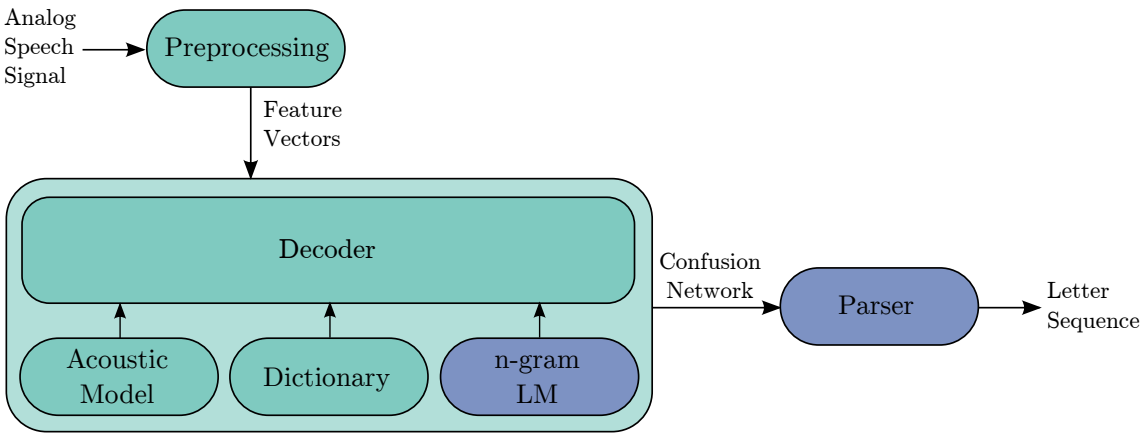

Figure 4.1: Block diagram of the Phrase Network System

The components highlighted in blue in the block diagram are the ones that are specially build for this system and for the task of spelling and thus differ from standard speech decoders. The system consists of two separate parts, which are invoked consecutively.

- The first part is the decoder itself, which is modified to output a confusion network instead of a hypothesis. The following Sections 4.2 and 4.3 explain its components in detail.
- The second part is a parser, which takes the confusion network and extracts information in form of phrases for spelling letters. From these phrases, the final sequence of spelled letters is then derived. The parser is the main work of implementation and is explained in Section 4.4.

A similar idea of splitting the process into two stages is presented in [BGWT04], as mentioned in Section 1.4.3. It has the advantage of combining the high recall of a confusion network with the high precision of grammars and exactly matching phrases.

The separation has the further advantage that the decoder can be seen as a black box. Any other decoder that outputs a suitable confusion network can be used instead. This makes it possible to test the parser with different speech recognition toolkits. However, in this work only the Janus decoder is used that is described in the next section. The two components are tuned to each other in order to work well together and thus to minimize errors. For example, the language model is trained on data similar to the sentences that the parser expects.

4.2 Decoder

The decoder component is a standard Janus-based system as described in Section 2.2 and Section 2.3. It is a continuous speech recognizer with acoustic models trained on the Wall Street Journal corpus, as they were already used and described in Section 3.3.1.

The advantage of using this field-tested system is that it is established in practice, with sophisticated and well-tested methods. Particularly the use of the n-gram based

language model in decoding is optimized for speed, as this is the state-of-the-art LM, which solves the speed problem of the CFG system (see Section 3.3.2).

Furthermore, being build for continuous speech, the recognizer is presumably better at decoding a sequence of phrases like the ones expected for the task of human-like spelling ("this is an upper case M as in michael") than single letter systems and neural networks approaches as presented in Section 1.4.1. However, there might a slight downside to it, because spoken single letters are usually pronounced independently ("M I C H A E L"), which is not what a continuous system expects. As word break tags `WB` (see Section 2.2.2) are used, this might though not be an issue at all.

The two differences compared to a standard system are the specialized LM and the output of confusion networks instead of hypotheses; these are the subjects of the following sections.

4.3 Language Model

Building an n-gram language model (see Section 2.2.3) for the use in a continuous speech recognizer is usually achieved by estimating the probabilities of all n-grams from a corpus of given text that is similar to the type of text expected in testing. In our case we had to work with sparse data, as it was not possible to collect an amount of suitable text that is sufficient for a reliable estimation of an LM. People tend to use the first word that comes to their mind to explain a letter, so "A as in axolotl" is a valid (yet unlikely) expression. This means that it is hardly feasible to collect a text corpus that contains every variant of valid expressions in a representative quantity.

To address this problem, the LM had to be generated from other sources. A possibility is to directly assign probabilities to the different n-grams, but in order to keep track of the details it is easier to generate a text corpus first and then estimate the n-gram LM from it.

4.3.1 Generating Text using a PCFG

As the structure of the wanted type of text is strict in terms of which kind of utterances are to be expected (see Section 3.2 for the spelling language), a CFG is a suitable way to model it. In order to also account for the probabilities of words used for spelling, a probabilistic component is needed, so that we used a PCFG to model the spelling language including the probabilities of production rules. We developed a small tool that generates random text according to a PCFG.

Section 1.4.2 presented some previous work on inferring formal grammars with their rules and probabilities from text corpora. In our case, this was not feasible, mainly because of data sparseness, but also because the extraction of information from non-handwritten rules entails further complexity.

Thus, the language described by the PCFG is derived from the spelling language presented in Section 3.2 and the same that was already used in the CFG system in Section 3.3.2; it can be found in Appendix B.1. It generates sentences consisting exclusively of phrases that the parser later on is able to process. All parts of a sentence that are not explicitly modeled in the parser would be omitted anyway during parsing, so this is a sufficient approach for training an LM for our purpose.

In addition to the language itself, a measure of probability had to be provided for each production rule. For the basic structure of the grammar, we used a rough estimation based on the collected data of Section 5.2. However, in case of the codewords for spelling letters ("J as in jaguar"), which are too many to collect robust data for, we had to use other sources of information to get reliable probabilities; this is described in the following Section 4.3.2. This was again a necessary consequence of the data sparseness problem. The amount of generated text had to be large enough to include at least the more common codewords with high certainty in order to provide reliable counts for estimating the LM.

The following example of the grammar shows four production rules, as used for our generator tool. The difference compared to the JSGF is that each line contains one rule only, without statements for optional or alternative productions, which made the implementation of the tool easier.

```
3    <case-modifier-all-lower>    lower case
3    <case-modifier-all-lower>    lower case letters
1    <case-modifier-all-lower>    small
1    <case-modifier-all-lower>    small letters
```

The numbers in the beginning are the weighting of the production rules, which indicate their probability. The distribution in this case is favoring the first two rules — which both have a probability of $\frac{3}{3+3+1+1} = 0.375$ — over the other rules with a probability of 0.125 each. Then follows the non-terminal of the rule in angle brackets and finally its productions, in this case only consisting of terminal symbols.

4.3.2 Estimating Word Probabilities

Using estimations of the production probabilities, which are based on the collected data, works well for the basic structure and recurring parts of the grammar like case modifiers and word intros. It however does not work well for the amount of words used in spelling phrases like "P as in pumpkin", because basically all words of the dictionary can be used here. Thus, it is not feasible to collect enough data for a reliable estimation of the probability of these phrases.

To get reasonable estimations, data from other sources had to be included. We used the following categories of word lists for this task.

Spelling Alphabets
: As using spelling alphabets to explain a letter like "G as in golf" is a basic mode of the recognizer, they are an important source to include in the LM. An exemplary list of some of the used alphabets and a full list of the sources can be found in Appendix A.3.

Entity Lists
: Most people do not know one of the classical spelling alphabets by heart and use other words that come to their mind as codes for a letter. According to our training data collection, these are frequently given names. We also included words of the following categories: nations, capital cities, major US cities, US states, US presidents and some more. In case of the given names, it was even possible to weight the lists by the commonness of the names [Butl02].

Google Ngrams

To cope with all the words that are not in one of the previously mentioned lists, an additional source of regular common words had to be used. Google Ngrams[11] [MSAV+11] is a large database of n-grams obtained from books. We used the "English 2012 1-gram corpus" that contains counts of millions of words derived from books predominantly in the English language. For each letter, the list was sorted by the counts of the words and up to 2,000 words were extracted and weighted using the counts.

Where applicable, an individual weight for their probability was assigned to the items of these lists, or equal weight otherwise. These lists were used to fill in for the placeholders like `@letterword-A` in the PCFG.

Each letter was resolved by production rules like the following:

```
15   <letter-A>   A
10   <letter-A>   A as in @letterword-A
 1   <letter-A>   A is for @letterword-A
 1   <letter-A>   A for @letterword-A
```

The cases of "A" and "A as in ..." are weighted more heavily than the two others, because they are more common. The frequency at which each placeholder like `@letterword-A` is replaced by a word of the lists is selected according to the weight of the word.

4.3.3 Estimating the LM

Given the language in form of a PCFG with the above mentioned lists of words to resolve the codewords for the letters, a text corpus was generated consisting of 1,000,000 sentences for each of the three categories of word lists. To create an individual corpus for each category makes it easier to handle the different amounts of words in each of them, as the weights of the words then only have to be consistent within each corpus, but not across them.

An exemplary utterance from the Google-Ngrams-generated corpus looks like this:

> "everything is lower case O as in otis B upper case T Q space Z small T as in tooth B I I as in indianapolis X double A as in airplane R Z E Y as in young"

The fact that this sentence does not make sense because of the contradiction between "everything is lower case" and "upper case T" is negligible, as the used n-gram context does not span this many words. We used a 4-gram model, because a context length of 4 is sufficient to cover phrases like "M as in mike", but still small enough to avoid over-fitting.

Also, for reasons of interpolation and smoothing, a text corpus from TED talks[12] consisting of about 270,000 sentences was used as a fourth text source. The idea was

[11] More information on this at https://books.google.com/ngrams/info (accessed 2014-01-10).
[12] Information available at http://www.ted.com/talks (accessed 2014-01-10).

to allow the users some inaccuracy and non-spelling language in their utterance to make it more flexible. As the parser is able to omit these parts when extracting spell phrases, it is thus better to explicitly model them in order to avoid false positives.

Using the SRILM toolkit[13] [Stol02], the four corpora were used to estimate four individual 4-gram LMs. This process is briefly introduced in Section 2.2.3. All LMs were then combined using the SRILM toolkit again to get one final 4-gram-LM to be used in the system.

4.4 Parser

This section discusses the parser that takes the output of the decoder and derives the sequence of letters being spelled in the utterance from it. It is the main part of the implementation of this work and was developed in C code as part of the Janus recognizer, making it a component to be used in the scripts after the decoder has done its work.

In this section, firstly the special type of confusion network we used is described, which serves as the link between the decoder and the parser. Then, the phrase structure is presented, which is used as the building blocks of spelling in the parser. Lastly, the actual parsing algorithm is explained, which does the main work of extracting phrases from a network.

4.4.1 Confusion Network

After the decoding is done, the system is configured to produce a confusion network, which contains the different hypotheses for the utterance in a compact format. For an introduction to confusion networks, see Section 2.2.4. We use them, because they are more robust against errors that occur in decoding than the 1-best hypothesis and thus are good for spontaneous speech; approaches of information extraction from lattices and confusion networks were also used in [SaSp04] and [HHHG07].

Usually, there are some fillers symbolizing breath and other human non-speech sounds, silence and noises in the network. These are filtered out when reading it into the parser, as they are not significant for the task of spelling. Figure 4.2 presents an example of a filtered confusion network with different phrases of spelled information; we here use the compact presentation form of word lists instead of a graph.

$$\begin{Bmatrix} \text{-} \\ \text{this} \\ \text{that} \end{Bmatrix} \begin{Bmatrix} \text{is} \\ \text{as} \end{Bmatrix} \begin{Bmatrix} \text{A} \\ \text{U} \end{Bmatrix} \begin{Bmatrix} \text{as} \\ \text{is} \end{Bmatrix} \begin{Bmatrix} \text{-} \\ \text{in} \end{Bmatrix} \begin{Bmatrix} \text{upper} \\ \text{apple} \end{Bmatrix} \begin{Bmatrix} \text{case} \\ \text{-} \end{Bmatrix} \begin{Bmatrix} \text{B} \\ \text{T} \end{Bmatrix} \begin{Bmatrix} \text{as} \\ \text{-} \end{Bmatrix} \begin{Bmatrix} \text{N} \\ \text{in} \\ \text{finn} \end{Bmatrix} \begin{Bmatrix} \text{toy} \\ \text{boy} \end{Bmatrix}$$

Figure 4.2: An exemplary confusion network containing spelled information

The "–" is a tag used to indicate that this node of the network can also be left out, which is a result when flattening the lattice into a confusion network. The decoder also outputs a score for each word in the network which represents the confidence of

[13] The project's website is http://www.speech.sri.com/projects/srilm (accessed 2014-01-10).

the decoder that this is the actual word being spoken in the utterance. These scores are not displayed in the figure.

This network contains several possible meaningful matches of spelled letter sequences that sometimes overlap with each other:

- “A as in apple”
- “U as in upper”
- “upper case B”
- “B as in boy”
- “upper case B as in boy”
- “upper case T”
- “T as in toy”
- “upper case T as in toy”
- the single letters “A”, “U”, “B”, “T” and “N”

The goal of the parser is then to extract these matches and deduce the sequence of letters from it. The confidence scores can be used to dissolve overlapping parts in order to find the best-scoring combination of phrases. To keep complexity low, these matches are split into smaller units of independent phrases, as presented in the next section.

4.4.2 Phrase Structure

The phrases to be extracted from the network are the ones presented in Section 3.2. In the parser, they are modeled as individual objects for each type of phrase and each length — e. g. one phrase object for the three-word sequences of “A like apple” and one for the four-word sequence “A as in apple”. Figure 4.3 shows such a phrase.

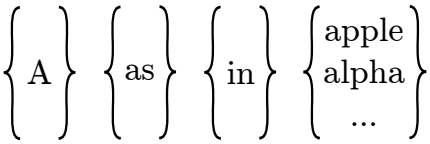

Figure 4.3: An example of a phrase for the letter A

The format of a phrase is essentially the same as the one used for the confusion network, as both represent a sequence of words with alternatives for words fitting at the same position. Each phrase object is provided with a special type determining its semantic meaning according to the spelling language. After extracting the phrases from the network, these types are used to generate the actual output of the parser. The phrases appear in the network in form of words in consecutive nodes.

In order to keep the implementation as flexible as possible, only the types of phrases and their semantic meaning are coded into the system, while the actual contents of the phrases are added via configuration scripts and input files. For example, the

phrase type “upper case letter” is realized by making the following letter upper case, whenever this type of phrase is detected in the network. But the actual words that translate into this phrase type — like “capitalized”, “capital” or “upper case” — are filled in when configuring the system at startup. This is mainly done by using lists of possible words representing the phrase. Also, more complex and variable content is possible by using wild-card and regular expressions e. g. to stand in for single letters or phrases like “A as in a. . . ”, making the system highly customizable. The complete startup script for loading the phrases can be found in Appendix B.2.

This also makes it possible to easily realize another language of spelling with the system. Our system is built and tested for spelling in English. If however spelling in another language is demanded, it may be necessary to implement some more types of phrases.

4.4.3 Parsing Algorithm

Provided with the set of phrase types and their actual content in form of word sequences, the matching algorithm extracts these phrases from a given confusion network. The algorithm is based on a **Depth First Search** (DFS) combined with pruning via a beam [ZhHa05]. The basic idea is to find intersections of the phrase graphs and the confusion network graph, thus it can be seen as a subgraph matching problem.

During the DFS, there are some special cases to be considered, hence there is need for some elaboration on the process. First, a general description of the algorithm is given, before details about its concrete implementation are presented, and finally the output generation is described that turns the found phrases into the final sequence of letters.

General Description

The DFS is an algorithm for traversing a graph data structure. The confusion network is a directed graph, where the nodes split the edges into segments for each period of time of the utterance. Figure 4.4 shows the graph for the confusion network of Figure 4.2; the symbols ⊙ and ◎ mark the beginning and end of the graph.

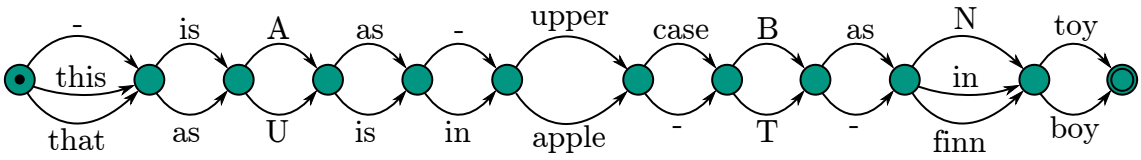

Figure 4.4: Confusion network displayed as a graph

In order to traverse the graph, the DFS first visits the nodes deeper in the graph by selecting the first edges every time until the end node is reached. The first path through the exemplary graph hence is

> “ - is A as - upper case B as N toy”

Then the algorithm goes back until an unvisited edge is available, and again continues until the end is reached — and so on. The next paths then are

“ - is A as - upper case B as N boy”
“ - is A as - upper case B as in toy”
“ - is A as - upper case B as in boy”
“ - is A as - upper case B as finn toy”
...

All these paths have to be examined for matching phrases and scored in order to find the best path. For example, the first path contains the phrases “A”, “upper case”, “B” and “N”, while another path occurring later in the execution of the algorithm has the phrases “A as in apple”, “upper case” and “T as in toy”, which covers more nodes and thus is probably a better candidate.

The complexity of a DFS for graphs of this kind is exponential in the depth of the graph. Hence, an exhaustive search is not always feasible and a strategy needs to be deployed to prevent too long execution times.

Every time the DFS enters a node in the graph, a partial score is calculated which expresses how ‘good’ the current path from the beginning of the graph to that node is, even if it is not the end node. The best scores are stored for each depth level of the graph. Whenever a new partial score is calculated for the path being explored during the execution of the algorithm, it is compared to the currently best score that leads to that level of depth in the graph. Thereby, when the current path is not good enough, its exploration can be stopped early, which is called **pruning** and saves time.

The assumption here is that a partial path that is known to be worse than another one that was explored and scored before, will not score better after the exploration of more edges deeper in the graph. However, to allow for some tolerance, a path is not cut off immediately once it scores worse than the currently best path to that depth. Instead, a **beam** is used as a factor applied to the best score: only if the current score is worse than the best score times the beam size, the corresponding path is stopped.

After the algorithm terminates, the scores for each level of depth represent the best paths to reach that level in the graph. Hence, the score of the end node represents the total best score and stands for the best path through the complete graph.

An advantage of the algorithm, particularly if real-time conditions have to be met, is that it is a so-called anytime algorithm. It finds a good, but likely sub-optimal solution quickly, then finds improved solutions given more time, until finally converging. This is because the edges are sorted by weight, so that the most probable ones are examined first. As the best path may however also contain other edges because of the constrictions imposed by the phrases, other paths still have to be visited. The execution of the algorithm then can be interrupted at any point, while it still delivers a good solution.

Although this is generally beneficial, in our tests (see Section 6.1.3) a time limit was rarely necessary, as the execution time is short enough when an appropriate beam size is chosen. Because the score of the optimal path has a relatively large distance to the score of other paths, a small beam of 1.01–1.20 proved to be sufficient, meaning that a path can be 1–20% worse than the currently best one before being stopped.

Concrete Implementation

Given that the graph of the confusion network is roughly as deep as the number of words in the utterance, the naive way of implementing a DFS by recursion is possible without risking a stack overflow. This makes the implementation easy and automatically saves information about the current step to the stack. The initial call of the function starts at the first segment of depth of the network, and every recursive call goes one segment deeper into it. The pseudo-code of the recursive function is listed in Algorithm 4.1.

Before running the DFS, two global arrays are initialized containing the currently best scores and the corresponding matches for every segment of the network. Figure 4.5 shows the structure of these arrays along with all possible phrases that are encountered during the execution of the algorithm.

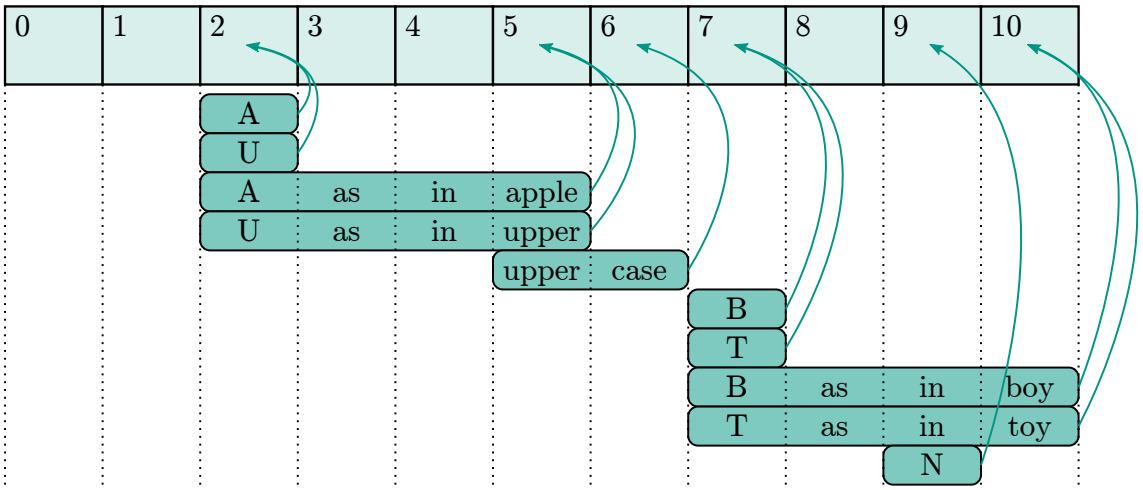

Figure 4.5: Score and match array of the DFS algorithm with all phrases

Whenever a phrase in found during the search, and it is better then the previously best phrase at this position, its score and the exact matching words are saved at the array position corresponding to the last edge of the phrase. This way, after the termination of the algorithm, the output generation can be achieved by iterating backwards through the array, which is explained later in this section.

The initial call of the recursive function is ParseDFS(0, 1.0, NULL, 0), starting at the first segment of the confusion network with the maximum score of 1.0 and no active phrases. The variables $activeMatch$ and $activePosition$ store whether in the current call stack there is a phrase being explored and at which position it currently is matching. For example, if in the last three calls the words “B as in” were matched, this is stored in these variables.

The first part of the function consists of the following break conditions which end the recursion and store the results if necessary:

- In lines 1 to 8, the case that currently no phrase is matching is handled. This happens when an edge of the network is treated as a filler, which causes a discount of factor $fillerPenalty$ in the score. Line 40 is the one that interprets a word as filler and calls the next recursion step. This makes it possible to skip words in the network, which was not possible with the CFG system (see

Algorithm 4.1 **ParseDFS** (int *currentNode*, float *currentScore*, Match *activeMatch*, int *activePosition*)

Input: global arrays: float *scores*[0..*n*], Match *matches*[0..*n*]

```
1  if no activeMatch then
2      if currentScore better than score[currentNode − 1] then
3          score[currentNode − 1] ← currentScore
4          matches[currentNode − 1] ← NULL
5      else if currentScore out of beam then
6          return
7      end if
8  end if

9  if activeMatch just finished then
10     currentScore ← currentScore · PhraseScore(activeMatch)
11     if currentScore better than score[currentNode − 1] then
12         score[currentNode − 1] ← currentScore
13         matches[currentNode − 1] ← activeMatch
14     else if currentScore out of beam then
15         return
16     end if
17     activeMatch ← NULL
18 end if

19 if end of network reached then
20     return
21 end if

22 for each word of network at currentNode do
23     newScore ← currentScore · Weight(word)
24     if word is the empty word "–" then
25         ParseDFS(currentNode+1, newScore, activeMatch, activePosition)
26     else if activeMatch then
27         if word matches activeMatch at activePosition then
28             activeMatch[activePosition] ← word
29             ParseDFS(currentNode + 1, newScore, activeMatch,
                        activePosition + 1)
30         else
31             continue
32         end if
33     else
34         for each phrase do
35             if word matches phrase at position 0 then
36                 create new match using word and phrase
37                 ParseDFS(currentNode + 1, newScore, match, 1)
38             end if
39         end for
40         ParseDFS(currentNode+1, currentScore·fillerPenalty, NULL, 0)
41     end if
42 end for
```

Section 3.3.2). If the score including the penalty is actually better than the currently best score leading to the current depth in the network, it is stored into the arrays. If the score is still within the beam, the examination of the current path is continued.

- Lines 9 to 18 deal with the case that a phrase was just finished with the last segment. Again, if the score is better than the currently best one, the phrase is stored into the arrays, which is also demonstrated in Figure 4.5. It is again checked whether the score is within the beam, and if so, the algorithm continues without an active match.
  The PhraseScore in line 10 is a plug-in hook for functions that score a completed phrase in order to weight it; so far, we only use it for the phrase mismatch correction, which is explained in Section 4.5.2.

- Finally, lines 19 to 21 check if there are still segments in the network left to be explored when the end node is not yet reached and break the recursion otherwise.

After these initial checks, the main loop follows. At this point of the execution, it is assured that the current score is within the beam size compared to the currently best path to this depth in the network. The loop, going from line 22 to line 42, iterates over the words of the current segment of the network.

In line 23, a new score is calculated by applying the weight of the word in the confusion network, which is a measure of the confidence of the decoder in this word. This step has primary influence on the score and thus is responsible for the distinction between 'good' paths and 'worse' ones.

The algorithm then continues to examine the current word by recursively calling the function with different parameters depending on the situation. There are four situations where a recursive call is made:

- Line 25 handles the case that the word currently being examined is the empty word, marked with the "–" tag. This means that this edge can be skipped without penalty; the function then simply moves on to the next node and segment without changing the active phrase.

- If there is an active match for a phrase being examined, it is checked whether the current word still matches to the next position of the phrase. If so, in line 29 a call is made to further pursue this match; if not, there is a mismatch between the word and the phrase, so there is no recursion call made then.

  For example, during the last three recursion steps, the partial phrase "A as in" has matched and thus is stored in $activeMatch$. If now the word currently being examined by the main loop is "apple", this matches the phrase at the current position $activePosition$, so a call to the next segment in the network is made.

- If however there is no active match being pursued in the current call, this means that the previous segment(s) in the network were interpreted as fillers, because they did not contain words matching any phrase. The algorithm then

tries to start a new one by iterating over all possible phrases. It tests whether the current word is matching the first position of a phrase, and if so, creates a new match and starts a recursion in line 37.

- After all phrases are tested, the algorithm also makes a recursion call in line 40, which is treating the current word as a filler. This is useful to allow the user to utter words that are not meant for spelling. A *fillerPenalty* is applied to the score, which makes sure that actual words are still preferred. This filler is not to be confused with the empty word, which essentially has the same effect of skipping a word, but without penalty, because it is a skip that is already planned by the network.

The combination of using the empty word and fillers to skip words in the confusion network makes the algorithm robust against non-spelling parts, because those parts without a matching phrase are simply left out. While the CFG system tries to find a possible derivation, even if there is none, the Phrase Network algorithm skips these parts if necessary.

An important characteristic of the score is that it has to be monotonous in order for the DFS to work correctly. This means, for each step the score either has to stay the same or has to decrease[14]. This way, the best path is the one whose score is closest to the start value of 1.0. As the weights of each segment along the path are multiplied to the score, the numbers approach zero, which may lead to numeric instability. Thus it makes sense to use a logarithmic scale for storing the scores.

In summary, the best path is mainly chosen depending on two criteria: Whether it contains phrases and what the weights of the contributing words are. The filler penalty and the phrase score are then used as additional criteria.

Output Generation

After the DFS algorithm terminates, the score and match arrays contain the best phrases that matched the utterance, as seen in Figure 4.6. Each position of the match array stores the best path to reach that position of the confusion network, which can either be a phrase (an arrow leads to this array element) or a filler (no arrow).

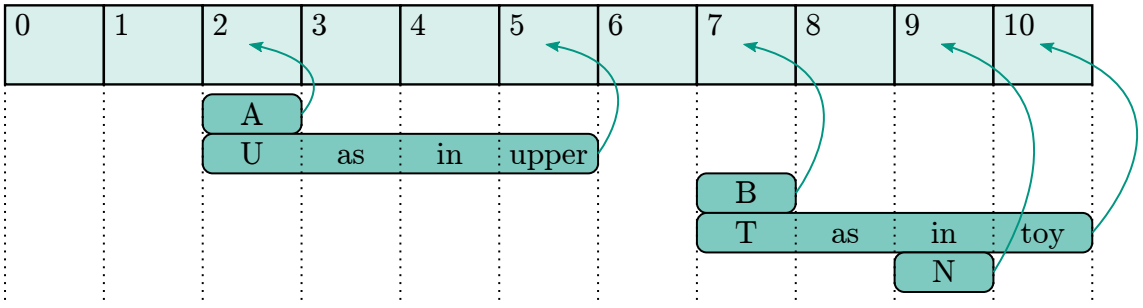

Figure 4.6: Score and match array containing only the final phrases after the DFS

In position 5, the phrase “U as in upper” is stored, which means that it scored better than the competing phrase “A as in apple”. The single letter “A” in position 2

[14] Increasing is also possible, as long as it is consistent.

scored better then the “U”, but is neglected, because it is covered by a longer phrase. Position 6 is empty, because the phrase “upper case” overlaps with “U as in upper”, which scored better, so that this position is treated as a filler. In positions 7 and 9, the letters “B” and “N” are stored, because these are the best paths leading to these positions. However, as the longer phrase “T as in toy” covers both, they are neglected.

The refusal of phrases that are overlapped or covered by longer phrases is possible, because the words of phrases are sufficiently disjoint to avoid collisions. This means for example that a phrase “A”, which overlaps with the phrase “A as in apple”, can be dropped, because no information is lost. There are however designed cases in which this might be a problem; to our knowledge, these did not appear in our data. If it turns out to be an issue in some cases, a different method for selecting the final phrases has to be chosen.

Given this array of best phrases, an extraction step then collects all phrases that are not refused. As the phrases are stored in the array element corresponding to their last word (indicated by the arrows in the figure), the extraction starts at the end of the array and iterates backwards. Whenever a phrase is found, it is extracted and the search continues at the array position in front of that phrase, until the beginning of the array is reached. This way, first the phrase “T as in toy”, covering positions 7–10, is extracted, then “U as in upper”, covering positions 2–5. Their original order however is maintained for the following step.

After the phrases are extracted, a last step transfers them into the final spelling sequence, which includes handling all phrase types as listed in Section 3.2. For capitalization, our parser supports different modes like all-upper-case, all-lower-case, first-upper etc, that are applied unless the speaker utters specifics on capitalization. A special automatic mode determines whether the spelled sequence is supposed to be a proper name by looking for phrases like “the name is. . . ” and proceeds accordingly by making the first letter of each word upper case. Also modifiers like “double” are applied. This step results in the final output sequence of letters.

4.5 Improvements

Having the parser decoupled from the decoder has the advantage of black box behavior, where the output of any decoder in form of a suitable confusion network can be used to extract spelling phrases from. This means that the parser cannot make specific assumption about the decoder but has to work with everything that it delivers. It is therefore desirable build in more linguistic knowledge and to restrict the decoder to those outputs that the parser can best handle. This is already partly achieved by the n-gram language model. A stricter language model like a context-free grammar would do this job better, but then again the parser would be needless. Thus, there has to be a compromise between restricting the decoder too much and too little.

As a consequence of this, inconsistencies and errors occur, so there is need for refinements. There are two scenarios that appeared in our tests which originate from this compromise:

- As single letters are allowed to be spelled individually, the decoder might output something like “F S N foxtrot” instead of “F as in foxtrot”.

- Although we use a 4-gram LM, which covers the context of all letter-related phrases, there might be a mismatch between the letter and its codeword like "B as in peter".

In order to cope with these scenarios, we developed two improvement methods, the confusion pairs and the phrase mismatch correction, which are presented in the following sections.

4.5.1 Confusion Pairs

As mentioned in Section 2.1.1, many letters of the English alphabet also sound like words. For training our LM, we mixed in a text corpus of continuous non-spelling speech (see Section 4.3.3), which means that these words can also result from decoding. In addition to cases such as "F as in foxtrot" leading to "F S N foxtrot", we observed confusions of "space" with "ace", "K as" with "case" and more.

This is not an issue as long as the corresponding correct words are also contained in the confusion network, because their phrases will score higher than the wrong sequences. But as our language model explicitly allows single letter sequences like "F S N", it sometimes drops the alternatives of "as" and "in".

A straightforward solution to this is to insert the missing words into the confusion network before running the parser. In order to have control over this process, we structured these words into weighted **confusion pairs**, where the occurrence of a word triggers the insertion of the other, using the weight as a discount for the word score in the confusion network. This discount is meant to not fully ignore the confidence that the decoder put into the decoded words. The discount was inferred from experience of working with the confusion networks, but as mentioned, correct phrases score higher anyway, so its influence is marginal. Table 4.1 shows the confusion pairs used for our main system.

Trigger word	Inserted word	Discount factor
S	as	1.0
N	in	1.0
you	U	1.0
axe	X	1.0
are	R	1.0
ace	space	0.8
is	as	0.8
of	F	0.8
and	N	0.6
an	N	0.6
case	K	0.5
case	as	0.5
has	as	0.5

Table 4.1: Confusion pairs and their discount factor

There are probably more pairs occurring in the data than the ones listed, and a proper estimation of the discount factors might be helpful, given more data. A possible data-driven approach is to collect the pairs from the comparison of hypotheses

and references. However, even with these few arbitrary confusion pairs, good results were achieved, as shown in Section 6.1.4.

As there are new words inserted into the confusion network, the probabilities of one segment do not longer sum up to 1. This is not an issue for our parsing algorithm, as the most probable path is searched independently from its exact score, but this should be kept in mind when using confusion pairs for other purposes.

4.5.2 Phrase Mismatch Correction

Although the context length of our 4-gram language model (see Section 4.3) covers phrases like "B as in boy", it is still a stochastic model that does not make hard decisions like a context-free grammar. Therefore, a mismatch of the letter and its codeword can occur, like "B as in peter", which happens when the decoder wrongly takes a 'P' for a 'B'. As we use the decoder as a black box, the parser has to cope with this. These errors occur because of the low acoustic distance between some letter names; a problem that is also discussed in [LoSp96].

In first tests, the phrases containing letters and codewords were built independently for each letter, with matching codewords only, as shown in Figure 4.3 in Section 4.4.2. In case of a mismatch, the parser then does not recognize the phrase. Thus the phrases have to contain all possible combinations of letters and codewords, in order for the parser to find them. Figure 4.7 shows such a phrase.

$$\begin{Bmatrix} \text{A} \\ \cdots \\ \text{Z} \end{Bmatrix} \quad \begin{Bmatrix} \text{as} \end{Bmatrix} \quad \begin{Bmatrix} \text{in} \end{Bmatrix} \quad \begin{Bmatrix} \text{apple} \\ \cdots \\ \text{zebra} \end{Bmatrix}$$

Figure 4.7: Phrase with all letters and codewords

After these phrases are identified in the confusion network, but before the output of the parser is generated, any occurring mismatches have to be corrected. In order to decide whether the letter or the codeword is correct, their confidences (weight in the confusion network) and their amount of acoustic evidence (length of the utterance section) are taken into account by multiplying them; the higher value is then chosen.

The motive to use the length as a factor is that codewords are used exactly because they have more acoustic evidence, so this should be considered. However, as finding the exact length of a word within the utterance requires information from the lattice (and not the given confusion network), we use an approximation instead: The parser is provided with the dictionary used in the system and looks up the number of phonemes that the letter and the codeword consist of. Thus, in the majority of cases, the codeword is longer, and — given that the weights of both are similar — is preferred in the decision process. Only in rare cases like "W as in wax", the letter is pronounced longer than the codeword.

A Supporting Feature

This basic approach copes with inconsistent phrases; however, the correction of mismatches between letters and their codewords is applied after the parsing is completed. It may thus be advantageous to already help the parser while the DFS

is running. This can be done by penalizing mismatches. In the description of the concrete implementation of the DFS algorithm in Section 4.4.3, the function PhraseScore is mentioned, which serves as a plug-in hook for functions that score a completed phrase. This is a suitable point to apply a penalty for mismatching phrases.

As explained in Section 2.1.1, some letter names like 'B' and 'P' have a small acoustic distance, while others like 'H' and 'W' have a higher one. As the phrases now allow all combinations of letters and codewords, there is a distinction to make between different mismatches which depends on the acoustic distance between a letter and the first letter of its codeword. For example, the phrase "B as in peter" might simply be a decoding mistake that can easily be corrected; however, the phrase "W as in peter" indicates that this phrase was not intended by the speaker at all. Therefore, the penalty should be higher in this case, and generally should depend on how 'bad' the mismatch is.

In the following paragraphs, we present a distance between letter names that is used to penalize mismatches. Its basic idea is as follows. First, an acoustically motivated phoneme distance is calculated from training data; this is then applied to the phoneme sequences of letter names to get a letter name distance; lastly, the mismatch scoring function uses the distance between letter names to penalize mismatches — a higher distance leads to a higher penalty. This function is the one that is then used for the PhraseScore function in the DFS algorithm.

Phoneme Distance

As we want the decision criterion for the letter name distance to be data driven and acoustically motivated, we use a vector distance function applied to the acoustic models of the phonemes as a starting point for the letter name distance function. For an introduction to acoustic modeling, see Section 2.2.1; a list of the used phonemes can be found in Appendix A.1.

The typical acoustic model in modern ASR systems uses **Gaussian Mixture Models** to approximate the distribution of a training set of feature vectors. For reasons of simplicity, in our phoneme distance function a GMM with just a single Gaussian is used as a rough but sufficient approximation for each phoneme. As using an acoustic model with one Gaussian only is atypical, it had to be generated from data first. Using audio data from the Quaero Project corpus [15], we extracted 200,000 randomly chosen samples for each phoneme and a GMM was estimated on these samples; this means the system had to be fully continuous to provide an independent GMM per phoneme.

Given these GMMs for all phonemes, the actual pairwise distance calculation was achieved using the **Extended Mahalanobis distance** [YKHL+98]. This distance measure adopts the Mahalanobis distance [Maha36] to Gaussian distributions $\mathcal{N}_1(x|\mu_1, \Sigma_1)$ and $\mathcal{N}_2(x|\mu_2, \Sigma_2)$ by using the means μ_i and the sum of their covariances Σ_i as parameters, as shown in Equation 4.1.

$$\mathrm{d}_{\mathrm{ExtMaha}}(\mathcal{N}_1, \mathcal{N}_2) = \sqrt{(\mu_2 - \mu_1)^{\mathrm{T}} \, (\Sigma_1 + \Sigma_2)^{-1} \, (\mu_2 - \mu_1)} \tag{4.1}$$

[15] See http://www.quaero.org for the project website (accessed 2014-02-10).

It can be seen as an extension to the Euclidean distance that takes the covariances into account and is therefore a suitable distance on Gaussians. This distance measure was then applied to the Gaussians of each pair of phonemes, which resulted in a phoneme distance matrix. The values had an arbitrary scale in a range of 10–50, and because for each calculation new sample vectors were extracted from data, even the distance between a phoneme and itself was greater than zero. A normalization step was not necessary at this point, because this is done after the calculation of letter name distances, which is described in the next paragraph.

Letter Name Distance

In this paragraph we define a distance function $\mathrm{LD}(l_1, l_2)$ between two letter names that represents how differently the pronunciation between them is. For example, the function is supposed to assign a low distance between 'F' (*ef*) and 'S' (*ess*), and a high distance between 'Q' (*cue*) and 'H' (*aitch*).

Therefore, the phoneme sequences of both letter names are aligned in time and the pairwise distances of aligned phonemes are added. Figure 4.8 shows two alignments with rounded distances.

	EH	F
EH	0	
S		15

(a) Distance of F and S

	K	Y	UW
EY	22	17	
		23	
CH		28	32

(b) Distance of Q and H

Figure 4.8: Letter name distance calculation

The alignment between 'F' and 'S' in Figure 4.8a is straightforward, because both letter names have the same number of phonemes. The distance between a phoneme and itself is set to zero, which also means that a letter name has a distance of zero to itself. The resulting distance between the phoneme sequences /EH F/ and /EH S/ is then $0 + 15 = 15$.

In case of phoneme sequences with different length, as in Figure 4.8b between 'Q' and 'H', the phonemes of the shorter sequence are interpolated to achieve an alignment with the longer one. In this example, the distance between /EY/ and /Y/ is 17, and between /CH/ and /Y/ is 28, which is interpolated to the average of 23, all values rounded. The resulting distance between the phoneme sequences /K Y UW/ and /EY CH/ is then $22 + 23 + 32 = 77$.

In an analogous manner, the distances between the phoneme sequences of all letter names are calculated[16]. A table providing the phoneme sequences can be found in Appendix A.2; a confusion matrix with the final distances for all letter names is presented in Appendix A.4. This matrix then contains the values for all combinations of letters for the function $\mathrm{LD}(l_1, l_2)$.

[16] The calculation is also possible for any two words by using a different dictionary.

The used phoneme distances are on an arbitrary scale and the phoneme sequences of different letters have different lengths, which means the number of added values differs. Therefore, the resulting distances should not be interpreted absolutely, but relatively to each other. We will normalize them in the next paragraph.

There are some interesting results to be found in the confusion matrix. For example, all letters of the E-set ('B', 'C', 'D', 'E', 'G', 'P', 'T', 'V' and 'Z') have small distances between each other, the same applies to the letters 'F', 'S' and 'X', while 'W' has a high distance to every other letter. This proves the plausibility of this calculation.

Mismatch Scoring Function

In order to turn the arbitrary scale of the letter name confusion matrix into a scoring function, we apply an exponential scaling. Given the distance function $\mathrm{LD}(l_1, l_2)$ between the names of two letters l_1 and l_2, the final scoring function is given in Equation 4.2.

$$\mathrm{PhraseScore}(l_1, l_2) = \exp\left(-\frac{1}{a} \cdot \mathrm{LD}(l_1, l_2)\right) \tag{4.2}$$

This score is used as plug-in for scoring a phrase during the DFS whenever a phrase consisting of a letter and a corresponding codeword is finished during search. It starts at a value of 1 for identical letters (no discount) and exponentially declines for higher distances. The scaling factor a is used to control the rate of decline; as the original distance values are in the range of $[0, 120]$ with an average of 50, we chose $a = 50$.

4.6 Live Feedback System

An important use case for spelling is OOV integration and error correction, as discussed in Section 3.1.1. But as errors also occur during spelling, a way of correcting them immediately by respelling might be helpful, particularly in hands-free environments that do not allow a keyboard or other means of input. In order to avoid respelling the whole sequence, we developed a system that asks for a specific letter and lets the user rephrase it. This process is triggered by a configurable confidence threshold, so whenever the decoder's confidence in a particular single letter is too low, it asks the user for repetition. This is achieved via a prompt on screen, as seen in Figure 4.9.

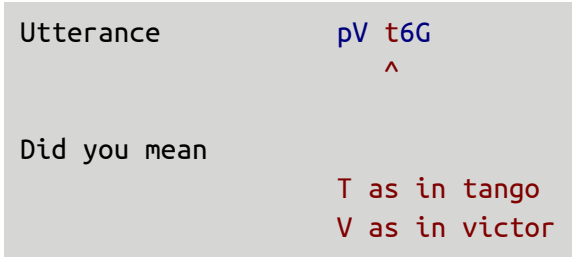

Figure 4.9: Prompt on screen of the live feedback system

The expressions suggested for the letter use codewords from the NATO alphabet. Apart from error correction, this also serves the purpose of accustoming the users to

these phrases; however they were not forced to use them, as the decoder accepts any form for spelling the letter. Spelling as a means of error correction was researched before, as presented in Section 1.4.4. Particularly [OgGo05] used confusion networks to present alternatives of letters with low confidence to the users on screen; in contrast to our system, users then selected the correct letter using a mouse or keyboard instead of restating it.

In order to test this feature, we used it in the collection of our test data, as described in Section 5.3. The users were presented a random string to be spelled, their utterance was instantly decoded using the full system presented in this chapter. If necessary, the feedback prompt was shown and the user was asked to respell a letter, which again was decoded to correct the error. We used a socket-based architecture to separate the user interface from the Janus decoder running in the background, which also makes it possible to use our spelling system as a network service.

4.7 Summary

In this chapter, we presented the Phrase Network system for identifying sequences of spelled letters and additional phrases used in human-like spelling. The system is based on a state-of-the-art continuous speech decoder with a language model specialized for the task of spelling. The actual information retrieval is achieved via an extraction and parsing step that works on the confusion network of the decoder by searching for spell-phrases using a depth-first search restricted by a beam for speed reasons. The system is improved with two methods. One that copes with letters that also sound like words and get easily confused by the decoder, and one that corrects mismatches between a letter and its codeword when being spelled. Furthermore, we presented a live feedback system, that allows users to interactively correct single letter errors in a spelling sequence.

5. Audio Data

The systems presented in the previous chapters are constructed to work on natural spelling, thus the test data needs to contain diverse and complex expressions. Most prefabricated data sets did not suffice for this specialized task, as they use artificial or restricted structures for spelling. There are a few sources of data available containing spelled single letters; Section 5.1 lists them. However, there were no audio sources available to us with sufficient sequences of spellings containing more complex structures. Therefore, we collected our own data — one set for training, which is the topic of Section 5.2, and one for testing, as described in Section 5.3.

5.1 Existing Data

A prominent dataset for spelled single letters is the ISOLET database[17], which consists of 7,800 spoken letters of the English alphabet spoken in isolation by 150 speakers, 75 male and 75 female, with each letter being produced twice per speaker, for approximately 75 minutes of speech. For more detail on the dataset see [CoMF90]. It is however not freely available and isolated letters are not a crucial part for both training and testing our system, so we did not acquire the ISOLET database.

Another dataset of spelled English proper names, given and last names like "G A R R E T T" or "W A L K E R", was available from previous experiments at the Interactive Systems Labs that consists of 2,000 utterances by 25 speakers for a total of 112 minutes of speech. It is recorded under quiet conditions, resulting in high quality speech. The advantage compared to the ISOLET dataset is that whole words were spelled, so we used it in suitable parts of the evaluation. However, it still uses single letters only; thus collecting our own data was unavoidable.

5.2 Collecting Training Data

In an early stage of developing the system, we collected training data with the focus on learning which ways of spelling people actually use to communicate complexly

[17] There are different web pages on this database, some of them are http://www.cslu.ogi.edu/corpora/isolet/, http://catalog.ldc.upenn.edu/LDC2008S07 and http://archive.ics.uci.edu/ml/datasets/ISOLET (all accessed 2014-01-22).

spelled words. The intended use of the outcome was to design the spelling language that the different systems use for decoding, see Section 3.2. Some of the tasks for the participants were similar to the ones used in [FaCR92]. In order to get real life utterances, the participants were hardly restricted in terms of the manner of spelling. The tasks were as follows.

1. **Spelling familiar terms**
   In the beginning, the participants are not biased towards a certain pattern of spelling, so the most natural results are to be expected. This it what a live system would probably have to cope with mostly. As most people have to do this occasionally, we expected that they have a particular habit of spelling their own name, thus the first task was to spell the given, any middle and the last name. For the same reason, the participants were asked to spell other words familiar to them, like street- and city-names, for example the ones where they live, or their pet's names.

2. **Spelling names and words**
   Some common names and other words on a list had to be spelled next. Exemplary utterances are "Angela Merkel", "Karlsruhe" and "Jibbigo", so mostly words that are not in the English language. Capitalization or the specification of spaces was not required.

3. **Spelling random strings**
   In this task, the participants were asked to spell (more or less) random strings in a way as creative as possible. Examples are: "alea iacta est", "BtCZec" and "Z3F64u". The goal was to get insight in some unusual spelling techniques when having to communicate passwords, passport numbers and akin sequences, where correctness of capitalization and spaces are important. Thus, the sequences contained upper and lower case letters, spaces and numbers.

4. **Creating a spelling alphabet**
   The participants were asked to come up with a spelling alphabet on their own: For each letter of the alphabet, they had to find a fitting word and record the phrase "A as in ...". This task was intended to find out which words are actually used by people, as most of the participants did not know the whole NATO alphabet.

5. **Reading spelled sequences using the NATO alphabet**
   The last task was to read sequences of words spelled using the NATO alphabet like "F as in foxtrot, O as in oscar, X as in x-ray". This was intended to deliver standardized comparable testing data.

This made a total of 90 spelling utterances per participant, which for reading convenience were given to them printed on paper. Furthermore, some background information was collected at the end of the sessions. The participants had to answer questions about their experience with ASR systems and possible improvements in order to get input on refinements of the recognizer. Some of the answers are analyzed in Section 6.3.3.

The recordings were conducted using and iPod Touch of the 4th Generation[18] and a recording software developed by Mobile Technologies, LLC[19]. The wave files are 16-bit Signed Integer PCM encoded with one channel at a sampling rate of 16 kHz.

The participants were sitting in an office room with mostly quiet environment; from time to time some background noise as well as traffic from outside occurred. The participants were native American English speakers mostly from Pittsburgh, PA, USA. 5 male and 7 female speakers in an age range between 23 and 62 with an average age of 36 years were recorded for a total of 178 minutes of speech.

In order to get analyzable output, two transcriptions had to be made for most utterances: One containing the actual content of the utterance like "D as in delta, O as in oscar, G as in golf" and one for the intended spelled word "dog". This task had to be done manually; we developed a tool for this, which is capable of some specifics of the task that standard transcription tools do not offer.

As the collected audio data itself was not used for training models of any kind, we also evaluated our systems on it for some additional insight, as is described in Chapter 6.

5.3 Collecting Test Data

After the development of the systems was completed, we collected more audio data for properly testing the performance of our systems and evaluating its accuracy.

This time, the collected data consisted of random sequences of letters separated into four categories as follows.

1. **Creative spelling**
   In the first task, the participants were asked to spell 25 sequences of 6 random letters each, using the structure of "A as in . . . " for each letter and filling in the gap with a fitting word that came to their minds. The idea was again that in the beginning there is no bias towards a special spelling alphabet when finding codewords to spell each letter, thus resulting in a natural way of selecting the words used.

2. **Spelling using the NATO alphabet**
   The second task was similar to the first, except this time the NATO alphabet was specified instead of free choice of words for the letters. Again, 25 sequences of 6 random letters each were recorded to produce utterances like "K as in kilo, . . . ". This data serves as a best-case baseline, because the 26 letters of the alphabet are encoded using a long, but consistent sequence of phonemes each.

3. **Spelling single letters**
   This task asked the participants simply to spell single letters like "A B C. . . " without any codewords. For this, the sequences from the first two tasks were used again (two times 25 random sequences of 6 letters each), which was intended for a direct comparison of the different ways of spelling in the evaluation. This is not needed for calculating the character error rate; as our evaluation mainly concentrates on the CER, we thus did not need these aligned

[18] Technical specifications available at http://support.apple.com/kb/sp594 (accessed 2014-01-10).
[19] The company's website is http://jibbigo.com (accessed 2014-01-10).

sequences in the end. They may however be useful for some further experiments.

4. **Random spelling using the live feedback system**
   The last task consisted of 20 sequences of upper and lower case letters, spaces and numbers, like "wd7kGj". The participants were free to use any way they could think of to make the system decode the utterances correctly, no restrictions were made on how to spell. This lead to utterances like "W lower case D as in dog the number seven K capital G as in girl J". It was intended for two purposes: Firstly, as test data for the main system as described in Chapter 4; secondly to test the live feedback system as described in Section 4.6. For this, the utterances were immediately decoded. Then, feedback for single letters with low decoding confidence was presented to the participants if necessary, asking them to respell a particular letter.

This resulted in a total of 120 utterances per participant, presented to them as prompts on a screen. The recordings were conducted using a Sennheiser E935 microphone[20] with a cardioid characteristic optimized for voice and speech and an Intel 82801H sound card[21] with the Linux ALSA driver 1.0.24[22]. The wave files are 16-bit Signed Integer PCM encoded with one channel at a sampling rate of 16 kHz.

The participants were again sitting in an office with some background and traffic noise; they were native American English speakers mostly from Pittsburgh, PA, USA. Except for one male speaker, none of them had been taking part in the training data collection of Section 5.2. There were 5 male and 6 female speakers in an age range between 22 and 37 and an average age of 31 years, recorded for a total of 222 minutes of speech.

This time, no manual transcription was necessary. For the tasks containing single letter sequences and spelling using the NATO-alphabet, the participants simply had to read from a prompt on the screen — this text was then used as reference; the first task with free choice of codewords also has little variation in it, and as the letters to spell were given on screen, a reference text in form of the sequence of letters also was available in this case. However, the exact words uttered by the participants (e. g. "F as in firefighter") are not known and transcribed — which is negligible, because in the evaluation it is more important which letter is finally outputted than which word was actually used to code it.

The data collected in this recording session is the main set used for evaluating our systems, which is the subject of the following chapter.

[20] Technical specifications are available at http://en-de.sennheiser.com/vocal-microphone-dynamic-cardioid-e-935 (accessed 2014-01-10).

[21] The data sheet is available at http://www.intel.com/design/chipsets/datashts/313056.htm and technical specifications at http://ark.intel.com/products/27684/Intel-82801H-IO-Controller (both accessed 2014-01-10).

[22] The ALSA project website is http://www.alsa-project.org (accessed 2014-01-10).

6. Evaluation

In this chapter, evaluations of the different systems of this work are presented. The basics of evaluation in speech recognition are introduced in Section 2.2.5. We will first present the system evaluations, including the two baseline systems and different variations of the phrase network system and its improvements. Then follows a system comparison, before finally some further findings are discussed.

6.1 System Evaluations

The evaluations were conducted using the **test dataset** of Section 5.3 mainly; were applicable, also the **training dataset** of Section 5.2 was used, as well as the **spelled proper names dataset** mentioned in Section 5.1. When systems with limited capability were evaluated, only suitable subsets of the whole datasets were used, which in each case is stated in the description of the experiment.

6.1.1 Single Letters

The single letter decoder as explained in Section 3.3.1 is a basic system that is only capable of recognizing sequences of single letters being spelled, like "P U M L U J". As most of our data contained more complex spelling, we only ran this system on the proper names dataset and a suitable subset of the test data. Table 6.1 shows the resulting character error rates (CERs).

Dataset	**CER**
Spelled proper names	26.2%
Test data: Single letters	57.0%

Table 6.1: Evaluation of the single letter system

The system achieved a character error rate of 26.2% on the spelled proper names dataset. Considering that the decoder was developed for continuous speech and only modified by using acoustic and language models for single letters, this is a decent result. This may be due to different factors. The quality of these recordings is high,

with clear speech and no background noises. Furthermore, the language model was trained on the Wall Street Journal corpus, which means it is biased towards the distribution of letters in standard English text, including proper names. Instead of an equal distribution, the most common letters in the dataset were 'A', 'E', 'R', 'N', 'I' and 'L', which made up more than half of the letters.

The second evaluation used the single letter subset of task 3 from the test dataset. As the utterances were purely random letters, the advantage of the biased language model did not work in this case, it might even haven been a disadvantage here. Thus, the system resulted in a CER of 57.0% on this dataset. Also, this data was not as clean and noise-free as the proper names set.

6.1.2 Context-Free Grammar

In Section 3.3.2, the system using a context-free grammar as language model is explained. It allows to decode utterances that use the full spelling language presented in Section 3.2. In Table 6.2, the system's performance on different subsets of the training and test data is shown.

Subset	CER
Names and words	74.8%
Random strings	76.4%
Create an alphabet	196.2%
NATO spelling	129.9%
Total	91.9%

(a) Training dataset

Subset	CER
Creative spelling	70.1%
NATO spelling	52.0%
Single letters	56.2%
Random strings	52.0%
Total	57.5%

(b) Test dataset

Table 6.2: Evaluation of the CFG system

In the evaluation on the training dataset in Table 6.2a, some of the problems mentioned in the system description emerge. For example, in the subset on names and words, one of the participants was asked to spell a random word that came to his mind, and he answered saying "okay and a random word is R A N D O M". The kind of non-spelling speech in the first part of this utterance is not provided by the CFG, thus errors occur.

Furthermore, short snippets that are modeled in the grammar are easily interpreted into an utterance without being there, because of the lack of control through probabilities. An example for this happened in the task on creating a spelling alphabet: One participant spelled "V as in victory" according to the task, but the system decoded "D S N victor eight", which then became "dsnv8" in the final output, instead of simply "v". This is the reason why there are two CERs above 100% — there are a lot of unnecessary letters inserted.

A further issue with the system on this dataset was the time needed for decoding some of the utterances. As the dataset contains many utterances with a lose spelling structure, the decoder had some trouble with them. Thus, we needed to stop 28% of the utterances of the training dataset during decoding due to a self-imposed time limit of one hour and had to ignore them in the evaluation.

The results on the test dataset in Table 6.2b were better, most probably because the utterances followed a stricter structure, which suits the CFG approach better.

However, there were still some letters inserted and the structure using codewords was not always correctly detected by the grammar.

Interestingly, these insertions also happened in the spelled proper name dataset, which reached a CER of 51.3% (this dataset is not listed in Table 6.2). In all three datasets, the letters that were mostly inserted are 'N', 'E' and 'R'. An explanation for the letter 'E' being inserted is that it is a lingering sound that follows all letter names of the E-set, thus e. g. the letter 'G', pronounced *gee*, becomes "G E".

This shows that a system for free spelling with a high variety of phrases based on a grammar is not a viable approach for most applications.

Restricted Grammars

As grammars are more appropriate for applications with a limited variety of utterances, we evaluated restricted grammars on suitable subsets of our test data. There are two ways a grammar for spelling can be restricted: First, by reducing the phrases; second, by predefining a length of the expected number of letters.

Our test dataset provided three subsets that use a strict structure for spelling:

1. The creative spelling task consists of utterances of the form "A as in ..." with any fitting word used as code for the letter.
2. The NATO spelling task further restricted this to using the NATO alphabet only to fill in as codewords. This means that for the 26 letters of the alphabet exactly 26 phrases of the form "A as in alpha" were used, which provides both a very limited set of possible phrases and a high amount of acoustic evidence per letter.
3. In the single letters task, no codewords were used, thus less evidence is provided. However, the space for inserting random letters is still there, which might again be a problem for the CFG.

We then created three grammars that only contain rules for these subsets, which were each evaluated on the respective subset. Furthermore, for all three grammars another variant was created that was restricted to a predefined number of letters being spelled. In our case this was 6 letters, because this is the length of all utterances in the test dataset. A variant of the full grammar limited to 6 letters was also evaluated on the whole test dataset. Table 6.3 shows the results.

	CER	
Subset	**Arbitrary length**	**Predefined length**
Creative spelling	16.8%	15.4%
NATO spelling	7.5%	1.5%
Single letters	53.6%	40.3%
Full grammar	57.5%	47.4%

Table 6.3: Evaluation of restricted CFG systems on subsets of the test data

The errors in the creative spelling task probably result again from the missing probabilities for the codewords in the grammar, so that some of the codewords were wrongly decoded to nonsense words.

Particularly good results were reached with the NATO spelling utterances, for the reasons stated above. This shows that the use of a grammar based system is feasible when enough restrictions are applied. In a multi user environment, this is not possible, but it is for applications like warehousing, where workers can be trained to use a specific spelling language. Furthermore, in this case the number of letters is known, because article numbers in the warehouse usually have the same length. The same applies to passports and other numbers. This makes a grammar based system a good choice in such cases.

Again, the single letters performed worst, because the limited grammar basically does not prevent the types of errors previously discussed, like random insertion of an 'E' after E-set letters.

6.1.3 Phrase Network

This section gives a detailed evaluation of our main system, which is described in Chapter 4. Unless otherwise declared, the systems use the following settings, which are explained in Section 4.4.3: a beam factor of 1.2, which means that paths that are 20% worse than the currently best path are stopped; a filler penalty of 0.2, which means that words that do not belong to a phrase are penalized with this factor. These values were taken because they proved to be adequate during the system development. Furthermore, both improvement methods are used, which are introduced in Section 4.5. Table 6.4 shows the results on different datasets.

Subset	CER
Names and words	40.0%
Random strings	38.9%
Create an alphabet	51.3%
NATO spelling	22.5%
Total	41.0%

(a) Training dataset

Subset	CER
Creative spelling	19.2%
NATO spelling	9.6%
Single letters	26.9%
Random strings	21.6%
Total	20.9%

(b) Test dataset

Table 6.4: Evaluation of the phrase network system

The evaluation on the spelled proper names dataset is not included in the tables; it scored a CER of 28.6%, which is comparable to the value of 26.9% on the single letter subset of the test data.

Again, the test dataset in Table 6.4b scored better than the training dataset in Table 6.4a, probably because it is cleaner and contains utterances with more structured spelling. It is striking that the variance of error rates on different systems is not nearly as high as in the CFG system. This means that the phrase network system delivers a steady performance on different tasks and thus is generally qualified for a broader range of applications — whereas the CFG system only helps in special scenarios.

Reduced Phrases

In Section 6.1.2, we presented an evaluation of the CFG system with grammars restricted to only some rules in order to narrow the search space in decoding down

and thus achieve better results. For reasons of completeness, we also evaluated the phrase network system with reduced phrases on suitable subsets of the test data. This means that the decoder still outputs a full confusion network, but the parser is only able to detect the limited amount phrases. This might be helpful to avoid insertions of single letters, like the E-set problem. However, a limitation to a certain length of the expected number of letters is not feasible with the phrase network system. Table 6.5 shows the results.

Subset	**CER**
Creative spelling	13.5%
NATO spelling	15.7%
Single letters	26.6%
Full phrases	20.9%

Table 6.5: Evaluation of restricted phrase network systems on subsets of the test data

The creative spelling task scored better than when using the full spelling language, probably because of single letters not being detected and thus not inserted where they are not supposed to be.

However, the NATO spelling task scored worse; this is because the confusion network contains codewords similar sounding to the NATO alphabet, like “kilos” instead of “kilo”, which were correctly parsed when all words were allowed in the phrases, but not when only the 26 NATO words are detectable. Furthermore, there is an issue with data pureness: for example, our list contains “whiskey” for the letter ‘W’, whereas the decoder also knows “whisky”, which caused errors. This is however an issue easy to fix by inserting variants of the NATO codewords to the list.

The single letter task was hardly affected by the limitation, because the utterances consisted of short letters only, so that the longer phrases of the full system were not detected in there anyway.

Capitalization

In some use cases, the correct capitalization of letters is important, for example in passwords. Table 6.6 shows the evaluation on two appropriate subsets, comparing the CER when the letter case is ignored to when it is considered. The random letter subsets are the only ones in our data where the participants had to include capitalization.

	CER	
Dataset	**Case insensitive**	**Case sensitive**
Training data: Random strings	38.9%	44.2%
Test data: Random strings	21.6%	23.6%

Table 6.6: Evaluation in consideration of capitalization

There are more errors in the case sensitive evaluation, because it is a further complication. However, the increase is not severe, particularly not on the test dataset.

Beam Size

The purpose of the beam is to provide a parameter for trading off accuracy against speed. Table 6.7 shows the effects of different beam sizes on the character error rate and the real time factor (RTF), which measures how long the parsing takes compared to the length of the utterance.

	Training data		Test data	
Beam	**CER**	**RTF**	**CER**	**RTF**
1.01	41.8%	0.03	20.9%	0.05
1.02	41.8%	0.03	20.9%	0.04
1.05	41.2%	0.05	20.9%	0.05
1.10	41.1%	0.07	20.9%	0.06
1.20	41.0%	0.12	20.9%	0.07
1.50	39.4%	0.19	20.8%	0.13

Table 6.7: Evaluation of different beam sizes

It is remarkable that the CER hardly improves with wider beams, which shows that the best path is found early during search, so that there is almost no effect of giving the system more time. This is a cause of the confusion network's edges being sorted by weight, so that the most probable ones are examined first, and makes the algorithm a good candidate for real live applications.

There is however an issue with the parsing duration and the RTF: some utterances needed too long for any reasonable application, so that the parsing had to be canceled. This happened more often with wider beams — in around 5% of the utterances for a beam of 1.5. We also tried wider beam sizes like 2.0 or 5.0, but parsing then becomes unfeasible slow, partially taking hours for one utterance. The cause are confusion networks coming from the decoder with many edges between two nodes that all have a low and similar weight, so that all of them are tried out during parsing. Because of that, the numbers in Table 6.7 are solely calculated on those utterances that resulted in an RTF < 2.00.

An easy solution to this problem is to use a narrow beam, which works, because the CER does not change much. Apart from that, a time limit for the parser could be implemented into the search. Because of its nature of being an anytime algorithm, as mentioned in Section 4.4.3, this is possible, while still getting a proper result.

The speedup on the test dataset between a beam size of 1.01 and 1.02 can be attributed to the degree of capacity utilization of the used computer cluster and is probably not a systematic anomaly. The default beam size of our systems is 1.20. According to these findings, for future experiments this can probably be changed to a beam more narrow in order to save time.

Filler Penalty

The filler penalty is included in the search to control how much a word is penalized if it is not used in a phrase but interpreted as a filler. The penalty is applied in form of a factor, so a low value means a high penalty and vice versa — a value of 1 does not penalize fillers at all. Table 6.8 shows the effect of different values on the performance with both datasets.

	CER	
Penalty	**Training data**	**Test data**
0.1	41.3%	21.1%
0.2	41.0%	20.7%
0.3	40.2%	20.9%
0.4	40.1%	21.0%
0.5	40.2%	21.5%
0.6	42.6%	22.9%
0.7	45.5%	24.2%
0.8	49.1%	26.5%
0.9	54.9%	30.9%
1.0	70.7%	52.5%

Table 6.8: Evaluation of different filler penalties

The first thing to notice is the increasing error rate with higher penalty factors, which means with less penalty for a filler. This may be due to the parser interpreting more and more words as fillers while they actually would fit into a phrase. This also explains the sudden jump at the limit of a penalty of 1.0, because in this case fillers are not penalized at all, thus the parser has no incentive to find any phrases.

This result also shows that our standard value of 0.2, which we estimated during development and achieved reasonable results with, is close to optimum.

6.1.4 Improvements

This section evaluates the impact of both improvement methods presented in Section 4.5. We check how the CER changes with deactivated confusion pairs and deactivated phrase mismatch correction.

Deactivated Confusion Pairs

Confusion pairs are introduced in Section 4.5.1 as a method to cope for letter and word homophones by inserting them to the confusion network before parsing. Table 6.9 proves that they actually decrease the character error rate.

	CER	
Subset	**Confusion pairs**	**No confusion pairs**
Creative spelling	19.2%	21.1%
NATO spelling	9.6%	9.8%
Single letters	26.9%	29.9%
Random strings	21.6%	24.0%
Total	20.9%	23.0%

Table 6.9: Evaluation of the effects of confusion pairs on the test data

The numbers in the left column are the same as in the evaluation of Table 6.4b, for reference; the right column shows how the CER rises when confusion pairs are not used. Particularly for the single letters test data they are an improvement, probably because of the extra letters that are inserted to form proper phrases. For example, without confusion pairs, the utterance “an ax are” results in an empty parse, but with them it becomes “N X R”, which is sound.

Deactivated Phrase Mismatch Correction

The phrase mismatch correction is introduced in Section 4.5.2 and is supposed to correct phrases coming from the decoder that have a mismatch between a letter and its codeword, like "V as in delta". It is applied after the search, when such a phrase is part of the final parse result. Furthermore, we presented a supporting feature to it that penalizes mismatches already during the search, so that they do not appear in the result in the first place. This feature is based on the acoustic distance between the letter name and the first letter of the codeword and gives a higher penalty for higher distances. Table 6.10 shows the results for both methods.

	CER		
Subset	**Correction**	**No penalty**	**No correction**
Creative spelling	19.2%	19.8%	18.6%
NATO spelling	9.6%	9.6%	11.7%
Single letters	26.9%	26.9%	26.7%
Random strings	21.6%	21.7%	21.1%
Total	20.9%	21.0%	21.0%

Table 6.10: Evaluation of the effects of phrase mismatch correction on the test data

Again, the numbers from Table 6.4b are shown for reference in the left column. The second column, labeled "No penalty", contains the CER when using the correction, but without applying the letter distance penalty first. The column labeled "No correction" shows the CER without the phrase mismatch correction, and thus also without the supporting penalty feature.

It is striking that the CER is in most cases even slightly better when the methods are not applied. This means, they do not properly work. Only in case of spelling using the NATO alphabet, the application of the mismatch correction improves the results, independently whether the penalty is also applied (left column) or not (middle column).

There are cases where the method actually works as intended, for example in one case, "R as in night" is not a correct phrase in itself, but when being corrected it becomes "N as in night". In other cases, it however backfired: For example, one participant uttered "E as in ear", which was decoded to "E as in your", and then became 'corrected' to be "Y as in your"; another one said "X as in xander", but with xander being an OOV in our system it became "X as in zander" and then "Z as in zander". It thus seems that the problem is more related to the decoding then to the parser component of the system. Both utterances worked properly without the correction, because phrases like "E as in your" are then simply parsed as the single letter 'E' followed by three filler words.

Another possible reason is explained in Section 6.3.2, where the accuracy of the confusion matrix is evaluated. In summary, the finding there is that the matrix may not be a good way to predict the confusability of letter names.

Our conclusion on this issue therefore is that the method works in theory, but may be not applicable in practice without some kind of coupling between decoder and

parser. This could avoid the mismatches in the first place, however it is not clear how such a coupling might work without using a grammar again. Furthermore, the letter distance matrix might need to be reworked.

6.1.5 Live Feedback System

The live feedback system is meant to immediately correct errors in the output by respelling the affected letter, as explained in Section 4.6. Table 6.11 shows the character error rate for those sequences that needed feedback, before and after the correction was applied. As explained in Section 5.3, these utterances solely come from the random letter subset of the test data, because this was the only part in the data collection where live decoding was used to collect feedback.

Feedback	CER
before application	26.8%
after application	15.2%

Table 6.11: Evaluation of the live feedback system

The result proves that the feedback system helps correcting errors. In our test data for this system, the feedback was needed in about 11% of the utterances, which means that the decoder had a relatively high confidence in the letters of all other utterances.

Furthermore, the live feedback introduced the participants to the use of the NATO alphabet, as mentioned in Section 4.6. An observation from the test data collection is that the participants initially tended to use single letters only, but later got accustomed to use phrases with codewords. Once they noticed that the feedback prompt uses these phrases and that they actually work better than single letters, they adapted to this pattern. This suggests that an indirect training of the users via suggestions can help improving the performance of a system.

6.2 System Comparison

The previous parts of this chapter presented evaluations of the different systems and their variations. In this section, we compare them to each other and draw conclusions for possible use cases.

Comparing the single letter baseline system to the phrase network system, we see that both score similarly on the spelled proper names dataset — 26.2% and 28.6% respectively — but that the latter achieved much better results on the single letter subset of the test data: 26.9% compared to 57.0% of the former system. This indicates that our system works better on random sequences and generally is more stable on different datasets.

A similar finding results from the comparison of the context-free grammar baseline system to the phrase network system. While the CFG system had a great variance in its CER depending on the used subset (see Table 6.2), our system again performed more equally (Table 6.4). Additionally, in comparison it got less than half the errors on most subsets. The grammar-based system is however better with restricted phrases on suitable subsets. For example, using the NATO alphabet for spelling

and a predefined length of the letter sequence, it achieved only 1.5% error rate. However, for most practical applications, this is too restricted to be used. This means, when allowing all phrases of our spelling language, the phrase network system also outperforms the CFG system by far.

However, the general performance of the phrase network system is somewhere between 20% and 40% CER, which is not accurate enough for many use cases. This may partly be due to the acoustic models of the decoder being trained on news data and not on spelling; given more data for our task, an adaption or even complete training might bring about an enhancement. Also, Section 7.3 discusses further ideas for improvements.

The evaluation of the details and parameters of the phrase network system showed that the additional phrases like modifiers for capitalization also work reasonably well. Furthermore, the filler penalty has a big influence on the error rate and a value of 0.2 proved to be good, which means that words being interpreted as filler get a discount of this factor. The beam size has little effect on the CER, thus a small beam of 1.01 can be used in order to speed the parsing up.

The two proposed improvement methods achieved differing results: While the confusion pairs increased the accuracy, the phrase mismatch correction has some issues in practice, that need to be solved before it actually has a positive effect on the quality of the system. The live feedback system further improved the performance and can in addition be used to introduce the use of spelling alphabets to the users.

A remark on the datasets: Throughout the evaluations, the recognition accuracy was better on the test dataset than on the training dataset. This can be ascribed to different factors. The intention of the training dataset was to find out which patterns of spelling people use, thus it contains more unstructured and random parts. In the collection of the test data however, the participants followed a stricter scheme, resulting in more structured utterances. This makes it easier for both the decoder and parser. Furthermore, the quality of the used recording device differed: for the training data, an iPod was used, whereas for the test data, a Sennheiser microphone optimized for speech was deployed, which reduced background noise and had generally a better quality.

6.3 Further Findings

In this section, some results are discussed that are not directly related to the system performances, but are interesting from a linguistic point of view and thus may further help improving the systems.

6.3.1 Used Code Words

As mentioned in Section 1.4.4, a user study to find a spelling alphabet for mobile devices was conducted in [LeCo03]. Using the task of creative spelling from the test data, we also evaluated a list of commonly used codewords. However, our list is not a design study in order to create a well working spelling alphabet, but instead results from what the participants actually used. Table 6.12 lists the three most commonly used codewords for each letter and their relative frequency.

	Common words					
Letter	**First**	**Freq.**	**Second**	**Freq.**	**Third**	**Freq.**
A	apple	25.7%	**alpha**	9.9%	all	5.9%
B	boy	36.5%	boat	6.3%	beta	6.3%
C	cat	28.1%	cats	15.8%	charles	7.0%
D	dog	29.0%	**delta**	7.2%	danny	5.8%
E	every	18.6%	elephant	16.3%	**echo**	14.0%
F	frank	25.5%	**foxtrot**	9.1%	france	9.1%
G	girl	31.2%	goat	12.5%	gate	6.2%
H	hank	8.8%	have	7.0%	human	7.0%
I	in	20.4%	ice	9.3%	irene	7.4%
J	jack	14.1%	job	8.5%	james	7.0%
K	kite	15.9%	kangaroo	14.3%	kate	12.7%
L	lamp	12.3%	lyon	7.0%	letter	7.0%
M	mary	23.2%	man	17.9%	mark	5.4%
N	nancy	25.0%	new	15.0%	night	8.8%
O	over	18.7%	oren	6.7%	old	6.7%
P	peter	31.3%	peanut	9.0%	**papa**	7.5%
Q	quiet	31.8%	queen	27.3%	quilts	7.6%
R	robert	16.7%	rabbit	9.3%	randi	5.6%
S	sam	19.2%	science	9.6%	stop	7.7%
T	thomas	15.5%	**tango**	12.7%	tree	9.9%
U	under	25.5%	unicorn	23.5%	umbrella	21.6%
V	**victor**	40.4%	very	8.8%	venison	8.8%
W	water	17.5%	william	14.3%	**whiskey**	12.7%
X	**x-ray**	56.9%	xylophone	27.5%	xerox	9.8%
Y	yes	22.1%	year	14.3%	yarn	7.8%
Z	zebra	57.5%	zander	11.0%	**zulu**	8.2%

Table 6.12: Most commonly used codewords in the creative spelling of the test data, with NATO words marked bold

Although it was originally intended for this purpose, we did not use the "Create a spelling alphabet"-part of the training data to create the table, because it provided only one word per letter per participant, which is not enough for proper evaluation. However, most of the words used there are in the given table as well.

There are several things to notice. The NATO alphabet words were only used for 10 letters, and only two times they were the most common ones ("victor" and "x-ray"); they are emphasized in the table. This means that the majority of users does not know or use this spelling alphabet. Thus, it is a good idea to also allow other codewords.

However, there are many particles and other short words in the list ("in/inn"[23], "new", "yes"), which makes it harder for the decoder. This is a contradiction to the idea that codewords are supposed to provide more acoustic evidence for a letter — for example the phrase "I as in in/inn" has more phonemes in the part "as in" than

[23] From the pronunciation, the two cannot be distinguished, so it is not clear which one the participant intended to use.

in the actual letter and codeword combined. This is probably a reason why the creative spelling task scored worse than the NATO alphabet task. Thus, in real live applications it might be helpful to introduce the users to a proper spelling alphabet, as was tried with the prompts in the live feedback system in Section 4.6.

Lastly, it is remarkable that some letters tend to accumulate one codeword — "zebra" was used in more than half the cases for the letter 'Z' — while others have a variety of words — the letter 'H' was explained using 32 different codewords in our data.

6.3.2 Letter Name Confusions

In Section 4.5.2, we described an algorithm to calculate the acoustic distance between two letter names, which resulted in a confusion matrix that can be found in Appendix A.4. This table can be compared to the letter confusions that actually happened in the decoding of the test data. For this, we calculated the Levenshtein alignment (see Section 2.2.5) between the references and hypotheses of the single letter task of the test data and evaluated which letters are commonly confused. Table 6.13 shows the five most common pairs and the frequency in how many cases that happened.

Reference	Hypothesis	Frequency
I	R	18.4%
M	N	15.5%
F	S	9.6%
H	A	9.2%
Z	E	8.7%

Table 6.13: Common letter confusions on the single letter task of the test data

All of these pairs also have a relatively low acoustic distance according to the confusion matrix. Merely the frequency of confusions of the pair 'H' and 'A' is higher than their distance predicts, when compared to other letter pairs with a similar distance. This may be due to the fact that the extra phoneme /CH/ of the letter name of 'H' is hushed when being spoken, so it is barely hearable, but fully counts in the distance calculation. Furthermore, it is striking how many pairs of letters have a low distance in the matrix, but are not often confused in the test data. It is however possible that more confusions would occur with more data.

This suggests that the confusion matrix is in general only an indirect indicator to predict the confusability of letter names in real data, but should not be interpreted as an absolute scale.

A related finding of the evaluation, particularly in cases where single letter spelling was used, is that often the letter 'E' is inserted in places after other letters of the E-set. This is because their sound ends with a long *ee*, so that for example the letter 'D' with the pronunciation *dee* easily becomes "D E". That problem also occurred in the CFG system, as mentioned in Section 6.1.2.

6.3.3 Background Information

During the collection of training data in Section 5.2, we asked the participants some questions about their experience with speech recognition systems and what they

expect from a spelling recognizer. This insight delivered some hints for features during our system development and is also interesting in general.

On the question how the participants like having to talk or even to spell to a computer, they showed mixed feelings: Some said, talking to a computer is awkward, while others learned to use it after an initial discomfort or even were totally fine from the beginning on, possibly because they have used such devices before. From their experience with speech recognition, some were frustrated of systems that did not even closely work or did not understand “yes” and “no”.

The participants also stated that they behave differently when spelling to a person and a computer: In the latter case, they try to annunciate clearly and slowly. Interestingly, some said that they would use single letters only, because they would not assume that the system is capable of more complex spelling expressions, while others on the contrary would give more examples like “B as in boy”, because there is less room for errors. A conclusion of this is that users need to know which utterances a system understands in order to use its full potential, thus giving them hints like we did in the live feedback system is helpful.

One participant said that a good device should have no need for spelling, unless it is a strange word — which basically refers to the use case of spelling for proper names, passwords and OOV words. Another one commented that people give immediate feedback, whereas a computer system usually waits for the utterance to complete before informing the user about errors. The user then does not know which part the computer did not get correctly and has to start over again, instead of being stopped mid-word or mid-sentence. We tried to solve this issue with the feedback system, where only those letters have to be respelled that caused errors.

7. Discussion

In this chapter, we discuss the results of this work. After a short summary of the work, conclusions are drawn, before finally an outlook is given and topics for future work are suggested.

7.1 Summary

In this work we presented a speech recognition system specialized on the detection of natural spelling using phrases like "T as in tango" as well as indicators for capitalization, spaces, numbers and some more. The system is based on a continuous speech recognizer which outputs a confusion network. This is then parsed for spelling phrases via a beam search, from which then the output is generated.

We also introduced two improvement methods that cope with issues that occur during decoding. Furthermore, a live feedback method was presented, that is able to immediately correct single letter errors by letting the user rephrase that letter.

This system is compared to two baseline systems: One is a single letter decoder, the other one uses a context-free grammar to restrict the search space to spelling phrases. Our system outperformed both of them.

7.2 Conclusions

The results from the evaluation prove that our system is better suited for most use cases than a single letter recognizer or a grammar-based approach. However, the overall performance of the system is somewhere between a character error rate of 20% and 40%, depending on the dataset. For many real live applications, this is not accurate enough.

Thus, this work should be seen as a feasibility study for the methods and approaches we proposed. The contribution of this work is therefore to present a system that is able to recognize human-like spelling, although it needs refinement and improvement.

Concluding from the tests with users during the recording session, particularly the task involving the live feedback system, we also see the challenge to let the users

know about the features of our system. If in a real life application still only single letters are used, because the user assumes that the system is not capable of more, than there is little gain in using such a system. Furthermore, as the usage of the NATO alphabet increases accuracy, users might need to be encouraged to use it instead of random words.

7.3 Outlook

One issue with the system is the decoding time measured in form of the real time factor, which was discussed in Section 6.1.3. An approach to solving the exponential dependency of the time from the depth of the confusion network is to first linearly extract all phrases contained in the network and then combining them to a sequence. This way, there is no need for a DFS on the full confusion network, but only on the actual combinations of phrases, which have a much smaller depth.

There is also some room for improving the accuracy. In some stages of the development we had to deal with sparse data, for example when training the language model — the acoustic model is not even trained on spelling data at all. Then again, more data is always the terminating resource in speech recognition, as summarized in the famous quote "There is no data like more data"[24].

But there are other approaches to improve the quality of a spelling recognizer. Instead of an n-gram language model, it might be helpful to use a phrase-based LM to model phrases like "O as in octopus" directly as a multi-word. In a similar manner, a multilevel approach is possible, where first the phrases are modeled as bigger units with a probability, which then consist of words that also follow certain probabilities, for example in order to emphasize the codewords.

For some applications, the n-gram probabilities for certain letters and letter sequences can also be adapted to the expected distribution, for example, the letters in proper names are not equally distributed. It might even be possible to use a background database to check the letter sequence, as was done in [FiSe04] and [ScRK00].

Also, the recognition could use a two-stage approach, where a first run detects the phrases and the basic structure of the utterance and a second run then uses a specialized dedicated single letter decoder on the relevant parts of the utterance to increase accuracy. It is also possible to conduct a speaker adaption on an utterance before the live feedback is invoked in order to boost the accuracy of the feedback utterance. Furthermore, as our parser assumes the decoder to be a black box, speech recognition systems can easily be combined to reach an agreement score between them.

An issue for many applications is the detection of spelled speech in an utterance that also contains normal continuous speech. [HiWa95] presents some methods for this. As our system uses the parser decoupled from the decoder, it is easy to implement a system for normal speech that additionally runs the parser downstream in order to check for spelling phrases. However, the decoder would then not be specialized for spelling, thus a second decoding might be needed on those parts of the utterance that the parser detects as spelling.

[24] According to [Jeli05], this comment was made by Bob Mercer at Arden House, 1985.

Finally, the selection of phrases in our work was limited to lists of words being used as code to elaborate on a letter. Using semantic analysis, it is also possible — though very complex — to introduce analogies in order to interpret phrases such as “meter like the measure”. This is one of the last steps needed to build a recognizer for fully understanding natural spelling.

A. Linguistic Resources

A.1 Phonemes

This table presents the phonemes used in our system and examples for them.

Phoneme	Example Words
AA	**a**rm, **a**rticle
AE	**a**venue, **a**xe
AH	**a**bout, **a**bove
AO	**awe**some, f**o**rce
AW	b**ou**nce, d**ow**n
AX	**a**ccount, **a**lert
AXR	capt**ure**, lit**er**
AY	m**i**ke, ps**y**cho
B	**b**rain, a**b**out
CH	**ch**ain, **ch**icken
D	**d**evelopment, **d**estiny
DH	**th**e, **th**ank
EH	**e**rror, **e**xcellent
ER	v**er**sus, t**er**m
EY	w**ei**ght, t**a**ke
F	**f**ilter, **f**lag
G	**g**old, **g**un
HH	**h**ack, **h**ammer
IH	h**i**story, **i**mage
IX	**i**llusion, **i**ntensive, ...**i**ng
IY	jewelle**y**, magaz**i**ne, ...t**y**
JH	ma**j**or, mer**g**e
K	mi**c**ro, **k**ill

Phoneme	Example Words
L	**l**ong, **l**ife
M	**m**an, **m**anual
N	**n**ovel, **n**ice
NG	la**ng**uage, ba**nk**, ...i**ng**
OW	b**o**ld, c**o**de
OY	app**oi**ntment, depl**oy**
P	**P**ittsburgh, **p**arty
R	**r**eason, **r**ecord
S	**s**enior, **s**etup
SH	**sh**ield, **sh**ort
T	**t**ime, **t**oday
TH	**th**umb, **th**eatre
UH	w**ou**ld, l**oo**k
UW	y**ou**, l**oo**se
V	o**v**er, pro**v**ider
W	q**u**een, **w**ay
XL	ab**le**, ang**le**
XM	rhyth**m**, touris**m**
XN	certai**n**, butto**n**
Y	**y**oung, **y**ear
Z	advi**ce**, i**s**, ...**s**
ZH	mea**s**ure, u**s**ual

Table A.1: Phonemes used in our systems with examples

A.2 English Alphabet

This is a list of the letters of the American English alphabet, with their most common pronunciation. The letter names are from the Oxford English Dictionary [Prof89]. The pronunciation is the one used in our system; an explanation of the used phonemes is provided in Appendix A.1. The frequency was calculated on the TED corpus as used in Section 4.3.3. Also some words are listed that sound like a letter.

Letter	Letter Name	Pronunciation	Frequency	Word
A	*a*	/EY/	8.153%	a (article)
B	*bee*	/B IY/	1.498%	be, bee
C	*cee*	/S IY/	2.685%	see, sea
D	*dee*	/D IY/	3.606%	
E	*e*	/IY/	12.085%	
F	*ef*	/EH F/	1.995%	
G	*gee*	/JH IY/	2.127%	
H	*aitch*	/EY CH/	5.460%	age
I	*i*	/AY/	7.368%	eye, I
J	*jay*	/JH EY/	0.157%	jay (bird)
K	*kay*	/K EY/	0.918%	
L	*el*	/EH L/	4.069%	
M	*em*	/EH M/	2.275%	
N	*en*	/EH N/	6.992%	
O	*o*	/OW/	7.974%	owe
P	*pee*	/P IY/	1.785%	pea
Q	*cue*	/K Y UW/	0.079%	cue, queue
R	*ar*	/AA R/	5.328%	are
S	*ess*	/EH S/	6.258%	as
T	*tee*	/T IY/	10.105%	tea
U	*u*	/Y UW/	3.040%	you
V	*vee*	/V IY/	1.071%	
W	*double-u*	/D AH B XL Y UW/ [25]	2.359%	double you
X	*ex*	/EH K S/	0.191%	ax/axe
Y	*wy*	/W AY/	2.326%	why
Z	*zee* [26]	/Z IY/	0.093%	

Table A.2: The English alphabet with its pronunciation

[25] This is the standard English pronunciation of the letter 'W'; there are also some common variations that our system supports.

[26] The letter 'Z' is pronounced *zee* in American English only, in most other dialects of English the pronunciation *zed* is used.

A.3 Spelling Alphabets

The following table shows some exemplary alphabets, which were used for generating the LM in Section 4.3.2. Furthermore, the NATO/ICAO alphabet was explicitly included in the spelling language in Section 3.2, which can also be seen in the grammar in Appendix B.1.

These and other alphabets used in the system were obtained from the following online sources: [ICA01], [Kelk07], [USF] and [Powe].

	NATO	**Western Union**	**U.S. Financial**	**U.S. Phonetic**
A	Alpha	Adams	Adam	Able
B	Bravo	Boston	Bob	Baker
C	Charlie	Chicago	Carol	Charlie
D	Delta	Denver	David	Dog
E	Echo	Easy	Eddie	Easy
F	Foxtrot	Frank	Frank	Fox
G	Golf	George	George	George
H	Hotel	Henry	Harry	How
I	India	Ida	Ike	Item
J	Juliet	John	Jim	Jig
K	Kilo	King	Kenny	King
L	Lima	Lincoln	Larry	Love
M	Mike	Mary	Mary	Mike
N	November	New-York	Nancy	Nan
O	Oscar	Ocean	Oliver	Oboe
P	Papa	Peter	Peter	Peter
Q	Quebec	Queen	Quincy	Queen
R	Romeo	Roger	Roger	Roger
S	Sierra	Sugar	Sam	Sugar
T	Tango	Thomas	Thomas	Tare
U	Uniform	Union	Uncle	Uncle
V	Victor	Victor	Vincent	Victor
W	Whiskey	William	William	William
X	X-ray	X-ray	Xavier	X-ray
Y	Yankee	Young	Yogi	Yoke
Z	Zulu	Zero	Zachary	Zebra

Table A.3: Some common spelling alphabets

A.4 Letter Name Confusion

This table shows the pairwise acoustic distances between the letter names when being spelled, as described in Section 4.5.2. The colors are for reading convenience only, and set at the thresholds of 30 and 45, where red marks the letter names that are easily confused and green the ones with higher acoustic distance.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	0	36	44	31	15	40	39	32	17	24	22	30	31	29	15	36	56	36	41	35	34	34	113	63	40	41
B	36	0	23	12	20	46	20	51	45	35	30	36	35	33	36	12	44	39	47	15	32	13	95	65	41	20
C	44	23	0	18	28	53	19	58	54	34	32	43	41	40	45	17	51	48	54	13	41	20	126	73	55	13
D	31	12	18	0	15	43	16	46	43	31	28	33	31	30	33	11	40	39	44	12	28	11	84	61	41	15
E	15	20	28	15	0	44	21	44	25	36	36	34	33	31	18	20	47	45	45	20	26	19	105	67	47	24
F	40	46	53	43	44	0	51	32	39	51	47	27	25	25	37	46	69	36	15	46	45	45	128	31	45	51
G	39	20	19	16	21	51	0	53	52	15	34	41	39	38	44	17	48	48	52	16	33	19	116	72	52	19
H	32	51	58	46	44	32	53	0	52	56	54	46	47	46	49	51	77	52	30	50	50	49	154	52	58	56
I	17	45	54	43	25	39	52	52	0	44	38	28	29	30	14	44	69	25	41	46	48	43	118	62	21	52
J	24	35	34	31	36	51	15	56	44	0	19	41	42	40	40	32	54	46	52	31	36	35	118	72	44	34
K	22	30	32	28	36	47	34	54	38	19	0	37	38	36	35	26	36	39	48	27	39	29	111	59	39	33
L	30	36	43	33	34	27	41	46	28	41	37	0	13	13	25	36	54	27	27	36	35	34	97	47	35	41
M	31	35	41	31	33	25	39	47	29	42	38	13	0	9	26	34	50	26	29	34	32	33	94	47	36	39
N	29	33	40	30	31	25	38	46	30	40	36	13	9	0	26	33	49	27	27	33	32	32	95	46	36	38
O	15	36	45	33	18	37	44	49	14	40	35	25	26	26	0	37	56	25	39	37	36	34	98	59	31	42
P	36	12	17	11	20	46	17	51	44	32	26	36	34	33	37	0	41	39	47	11	33	12	101	63	43	17
Q	56	44	51	40	47	69	48	77	69	54	36	54	50	49	56	41	0	62	70	42	29	43	78	69	68	50
R	36	39	48	39	45	36	48	52	25	46	39	27	26	27	25	39	62	0	39	40	44	38	107	59	30	47
S	41	47	54	44	45	15	52	30	41	52	48	27	29	27	39	47	70	39	0	46	47	45	131	18	48	51
T	35	15	13	12	20	46	16	50	46	31	27	36	34	33	37	11	42	40	46	0	33	14	104	64	46	14
U	34	32	41	28	26	45	33	50	48	36	39	35	32	32	36	33	29	44	47	33	0	32	77	68	43	38
V	34	13	20	11	19	45	19	49	43	34	29	34	33	32	34	12	43	38	45	14	32	0	100	63	43	17
W	113	95	126	84	105	128	116	154	118	118	111	97	94	95	98	101	78	107	131	104	77	100	0	130	110	117
X	63	65	73	61	67	31	72	52	62	72	59	47	47	46	59	63	69	59	18	64	68	63	130	0	69	71
Y	40	41	55	41	47	45	52	58	21	44	39	35	36	36	31	43	68	30	48	46	43	43	110	69	0	51
Z	41	20	13	15	24	51	19	56	52	34	33	41	39	38	42	17	50	47	51	14	38	17	117	71	51	0

Table A.4: Confusion matrix for letter names

B. System Resources

B.1 Spelling Grammar

The following grammar in JSGF notation is used in the CFG decoder in Section 3.3.2 and, in an adapted format, for generating the LM in Section 4.3.

```
#JSGF V1.0 ASCII;
grammar spelling;

//============================================================================
//  Utterance
//============================================================================

// this is the start utterance, everything else is integrated through this
public <utterance> =
    [<utt-intro>]
    [<case-modifier-all>]
    <word> +
;

<utt-intro> =
    ((this | that) [one] is | thats | its | it is)
    [<cardinal-digit> (word | words)]
;

//============================================================================
//  Modifiers
//============================================================================

<case-modifier-all> =
    <case-modifier-all-upper> | <case-modifier-all-lower>
;

<case-modifier-all-upper> =
    (all | everything) [of (these | them | this)] [is | are] [in]
    (capitalized | capitalised | capitals | capital | upper case | uppercase)
    [letters]
;
```

```
<case-modifier-all-lower> =
    (all | everything) [of (these | them | this)] [is | are] [in]
    (lower case | lowercase | small)
    [letters]
;

<case-modifier-upper> =
    capitalized | capitalised | capitals | capital | upper case | uppercase
;

<case-modifier-lower> =
    lower case | lowercase | small
;

<double> =
    double
;

//============================================================================
//  Words
//============================================================================

<word> =
    [<word-intro> | <name-intro>]
    <char> +
;

<word-intro> =
    [the] (<ordinal-digit> | next | last) word [is] | space | blank
;

<name-intro> =
    [the | my] (<ordinal-digit> | next | given | middle | last | family)
    name [is] [spelled] | (the | my) name [is] [spelled]
;

<char> =
    <letter> | [[the] number] <number>
;

//============================================================================
//  Letters
//============================================================================

<letter> =
    [<double>]
    [<case-modifier-lower> | <case-modifier-upper>]
    (
        <letter-A> | <letter-B> | <letter-C> | <letter-D> | <letter-E> |
        <letter-F> | <letter-G> | <letter-H> | <letter-I> | <letter-J> |
        <letter-K> | <letter-L> | <letter-M> | <letter-N> | <letter-O> |
        <letter-P> | <letter-Q> | <letter-R> | <letter-S> | <letter-T> |
        <letter-U> | <letter-V> | <letter-W> | <letter-X> | <letter-Y> |
        <letter-Z>
    )
;
```

```
<letter-A> = A | alpha    | A (as in | like in | like | is for) @letterword-A;
<letter-B> = B | bravo    | B (as in | like in | like | is for) @letterword-B;
<letter-C> = C | charlie  | C (as in | like in | like | is for) @letterword-C;
<letter-D> = D | delta    | D (as in | like in | like | is for) @letterword-D;
<letter-E> = E | echo     | E (as in | like in | like | is for) @letterword-E;
<letter-F> = F | foxtrot  | F (as in | like in | like | is for) @letterword-F;
<letter-G> = G | golf     | G (as in | like in | like | is for) @letterword-G;
<letter-H> = H | hotel    | H (as in | like in | like | is for) @letterword-H;
<letter-I> = I | india    | I (as in | like in | like | is for) @letterword-I;
<letter-J> = J | juliet   | J (as in | like in | like | is for) @letterword-J;
<letter-K> = K | kilo     | K (as in | like in | like | is for) @letterword-K;
<letter-L> = L | lima     | L (as in | like in | like | is for) @letterword-L;
<letter-M> = M | mike     | M (as in | like in | like | is for) @letterword-M;
<letter-N> = N | november | N (as in | like in | like | is for) @letterword-N;
<letter-O> = O | oscar    | O (as in | like in | like | is for) @letterword-O;
<letter-P> = P | papa     | P (as in | like in | like | is for) @letterword-P;
<letter-Q> = Q | quebec   | Q (as in | like in | like | is for) @letterword-Q;
<letter-R> = R | romeo    | R (as in | like in | like | is for) @letterword-R;
<letter-S> = S | sierra   | S (as in | like in | like | is for) @letterword-S;
<letter-T> = T | tango    | T (as in | like in | like | is for) @letterword-T;
<letter-U> = U | uniform  | U (as in | like in | like | is for) @letterword-U;
<letter-V> = V | victor   | V (as in | like in | like | is for) @letterword-V;
<letter-W> = W | whiskey  | W (as in | like in | like | is for) @letterword-W;
<letter-X> = X | x-ray    | X (as in | like in | like | is for) @letterword-X;
<letter-Y> = Y | yankee   | Y (as in | like in | like | is for) @letterword-Y;
<letter-Z> = Z | zulu     | Z (as in | like in | like | is for) @letterword-Z;

//============================================================================
//  Numbers
//============================================================================

// we use single rules for that, as it makes parsing convenient
<number> = <number-0> | <number-1> | <number-2> | <number-3> | <number-4>
         | <number-5> | <number-6> | <number-7> | <number-8> | <number-9>;

<number-0> = zero;
<number-1> = one;
<number-2> = two;
<number-3> = three;
<number-4> = four;
<number-5> = five;
<number-6> = six;
<number-7> = seven;
<number-8> = eight;
<number-9> = nine;

//============================================================================
//  Helpers
//============================================================================

<cardinal-digit>      = one   | two     | three  | four   | five
                      | six   | seven   | eight  | nine;
<ordinal-digit>       = first | second  | third  | fourth | fifth
                      | sixth | seventh | eighth | ninth;
```

B.2 Phrase Network Configuration

The following Tcl script is used to initialize the parser component of our system in Janus, as described in Section 4.4. The backslash \ marks lines that had to be wrapped for printing.

```
# ================================================================================
# Prepare Spelling Objects
# ================================================================================

# create main object
Spell spell$SID
spell$SID lsread "spell/letter-distance-scores" -scale 50
spell$SID configure -dict dict$SID

# configure confusion network object
spell$SID.net configure -fillerWords "<unk> <s> </s>"
spell$SID.net readcp "spell/confusion-pairs"
puts "spell$SID.net:    [spell$SID.net configure]"

# configure the parser object
spell$SID.parser configure -defaultCase "auto" -beamFactor 1.2 -fillerPen 0.2
puts "spell$SID.parser: [spell$SID.parser configure]"

# ================================================================================
# Create Phrases
# ================================================================================

# --------------------------------
# single letters
# --------------------------------

spell$SID.phrases create "letter" 1 -type "single_letter"
set sl spell$SID.phrases:"letter"
$sl.node(0) add  "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"

# --------------------------------
# single words
# --------------------------------

spell$SID.phrases create "word" 1 -type "single_word"
set lw spell$SID.phrases:"word"
$lw.node(0) read list "spell/nato.v"

# --------------------------------
# letter word
# --------------------------------

# this was not used in the final system, it is here for reference only
spell$SID.phrases create "letter_word" 2 -type "letter_word"
set lw spell$SID.phrases:"letter_word"

$lw.node(0) add  "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
$lw.node(1) configure -wildcard "*"
```

```
# --------------------------------
# letter for/like word
# --------------------------------

spell$SID.phrases create "letter_x_word" 3 -type "letter_x_word"
set lw spell$SID.phrases:"letter_x_word"

$lw.node(0) add  "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
$lw.node(1) add  "for like"
$lw.node(2) configure -wildcard "*"

# --------------------------------
# letter is/stands for word
# --------------------------------

spell$SID.phrases create "letter_x_for_word" 4 -type "letter_x_x_word"
set lw spell$SID.phrases:"letter_x_for_word"

$lw.node(0) add  "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
$lw.node(1) add  "is stands"
$lw.node(2) add  "for"
$lw.node(3) configure -wildcard "*"

# --------------------------------
# letter as/like in word
# --------------------------------

spell$SID.phrases create "letter_x_in_word" 4 -type "letter_x_x_word"
set lw spell$SID.phrases:"letter_x_in_word"

$lw.node(0) add  "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
$lw.node(1) add  "as like"
$lw.node(2) add  "in"
$lw.node(3) configure -wildcard "*"

# --------------------------------
# word/name intros
# --------------------------------

spell$SID.phrases add "wordintro-1"  "{space blank}"                          \
        -type "word_intro"
spell$SID.phrases add "wordintro-2"  "{the first next last}   {word}"         \
        -type "word_intro"
spell$SID.phrases add "wordintro-3"  "{the first next last}   {word}          \
        {is}" -type "word_intro"
spell$SID.phrases add "wordintro-4"  "{the} {first next last} {word}          \
        {is}" -type "word_intro"

spell$SID.phrases add "nameintro-2"  "{the my   first next given middle       \
        last family} {name}"      -type "name_intro"
spell$SID.phrases add "nameintro-3a" "{the my} {first next given middle       \
        last family} {name}"      -type "name_intro"

spell$SID.phrases add "nameintro-3b" "{the my   first next given middle       \
        last family} {name} {is}" -type "name_intro"
spell$SID.phrases add "nameintro-4a" "{the my} {first next given middle       \
        last family} {name} {is}" -type "name_intro"
```

```
spell$SID.phrases add "nameintro-4b" "{the my   first next given middle          \
        last family} {name} {is} {spelled}" -type "name_intro"
spell$SID.phrases:"nameintro-4b".node(3) configure -wildcard "*"

spell$SID.phrases add "nameintro-5"  "{the my} {first next given middle          \
        last family} {name} {is} {spelled}" -type "name_intro"
spell$SID.phrases:"nameintro-5".node(4) configure -wildcard "*"

# --------------------------------
# modifiers
# --------------------------------

spell$SID.phrases read "spell/mod-all-upper" "mod_all_upper" -type "mod_all_upper"
spell$SID.phrases read "spell/mod-all-lower" "mod_all_lower" -type "mod_all_lower"

spell$SID.phrases add "mod_letter_upper-1" "{capitalized capitalised             \
        capital uppercase}" -type "mod_letter_upper"
spell$SID.phrases add "mod_letter_upper-2" "upper case" -type "mod_letter_upper"

spell$SID.phrases add "mod_letter_lower-1" "{lowercase small}"                   \
        -type "mod_letter_lower"
spell$SID.phrases add "mod_letter_lower-2" "lower case"                          \
        -type "mod_letter_lower"

spell$SID.phrases add "double" "double" -type "mod_double"

# --------------------------------
# numbers
# --------------------------------

# mapping from number words to int values
spell$SID.parser numread "spell/numerals.m"

# phrases like "six"
spell$SID.phrases create "number-1" 1 -type "number"
set nw spell$SID.phrases:"number-1"
$nw.node(0) read list "spell/numerals.v"

# phrases like "number six"
spell$SID.phrases create "number-2" 2 -type "number"
set nw spell$SID.phrases:"number-2"
$nw.node(0) add "number"
$nw.node(1) read list "spell/numerals.v"

# phrases like "the number six"
spell$SID.phrases create "number-3" 3 -type "number"
set nw spell$SID.phrases:"number-3"
$nw.node(0) add "the"
$nw.node(1) add "number"
$nw.node(2) read list "spell/numerals.v"
```

Bibliography

[BaJu99] J. G. Bauer and J. Junkawitsch. Accurate recognition of city names with spelling as a fall back strategy. In *Proceedings of the EUROSPEECH Conference*, 1999.

[BeHi95] M. Betz and H. Hild. Language models for a spelled letter recognizer. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 1. IEEE, 1995, p. 856–859.

[Bell04] J. R. Bellegarda. Statistical language model adaptation: review and perspectives. *Speech communication* 42(1), 2004, p. 93–108.

[BGWT04] F. Béchet, A. L. Gorin, J. H. Wright and D. H. Tür. Detecting and extracting named entities from spontaneous speech in a mixed-initiative spoken dialogue context: How May I Help You? *Speech Communication* 42(2), 2004, p. 207–225.

[Butl02] R. A. Butler. Most Common First Names and Last Names in the U.S., 2002. http://names.mongabay.com/ (accessed 2014-02-20).

[CaOn94] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In R. C. Carrasco and J. Oncina (Ed.), *Grammatical Inference and Applications*, Volume 862 of *Lecture Notes in Computer Science*, p. 139–152. Springer Berlin Heidelberg, 1994.

[CFGJ91] R. A. Cole, M. Fanty, M. Gopalakrishnan and R. D. Janssen. Speaker-independent name retrieval from spellings using a database of 50000 names. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*. IEEE, 1991, p. 325–328.

[CFMG90] R. A. Cole, M. Fanty, Y. Muthusamy and M. Gopalakrishnan. Speaker-independent recognition of spoken English letters. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN*. IEEE, 1990, p. 45–51.

[Chom56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory* 2(3), 1956, p. 113–124.

[Chom57] N. Chomsky. *Syntactic Structures*. Mouton classic. 1957.

[ChSW03] G. Chung, S. Seneff and C. Wang. Automatic Acquisition of Names Using Speak and Spell Mode in Spoken Dialogue Systems. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, NAACL*, Volume 1, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics, p. 32–39.

[CiKr03] O. Cicchello and S. C. Kremer. Inducing grammars from sparse data sets: a survey of algorithms and results. *The Journal of Machine Learning Research* Volume 4, 2003, p. 603–632.

[CoFa90] R. Cole and M. Fanty. Spoken Letter Recognition. In *Proceedings of the Third DARPA Speech and Natural Language Workshop*, 1990, p. 385–390.

[CoMF90] R. Cole, Y. Muthusamy and M. Fanty. *The ISOLET Spoken Letter Database*. Technical Report. Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering. 1990.

[CoSL90] R. A. Cole, R. M. Stern and M. J. Lasry. Readings in Speech Recognition. Kapitel Performing Fine Phonetic Distinctions: Templates Versus Features, p. 214–224. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[CSPB+83] R. A. Cole, R. M. Stern, M. Phillips, S. Brill, A. Pilant and P. Specker. Feature-based speaker-independent recognition of isolated English letters. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 8. IEEE, 1983, p. 731–733.

[DaBr96] P. T. Daniels and W. Bright. *The world's writing systems*. Oxford University Press. 1996.

[EJLS90] S. Euler, B.-H. Juang, C.-H. Lee and F. Soong. Statistical segmentation and word modeling techniques in isolated word recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*. IEEE, 1990, p. 745–748.

[FaCR92] M. Fanty, R. A. Cole and K. Roginski. English alphabet recognition with telephone speech. In *Advances in neural information processing systems 4*, 1992.

[FGHK+97] M. Finke, P. Geutner, H. Hild, T. Kemp, K. Ries and M. Westphal. The Karlsruhe-Verbmobil speech recognition engine. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 1. IEEE, Apr 1997, p. 83–86.

[FiSe04] E. Filisko and S. Seneff. Error detection and recovery in spoken dialogue systems. In *Proceedings of the North American Chapter of the Association for Computational Linguistics on Human Language Technology 2004 Workshop on Spoken Language Understanding for Conversational Systems, HLT-NAACL*, 2004, p. 31–38.

[Gris97] R. Grishman. Information extraction: Techniques and challenges. In *Information Extraction A Multidisciplinary Approach to an Emerging Information Technology*, p. 10–27. Springer, 1997.

[Gris98] R. Grishman. Information extraction and speech recognition. In *Proceedings of the Broadcast News Transcription and Understanding Workshop*, 1998, p. 159–165.

[HHHG07] T. Hori, I. L. Hetherington, T. J. Hazen and J. R. Glass. Open-vocabulary spoken utterance retrieval using confusion networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 4. IEEE, 2007, p. 73–76.

[HiWa93] H. Hild and A. Waibel. Speaker-independent connected letter recognition with a multi-state time delay neural network. In *Proceedings of the EUROSPEECH Conference*, Volume 93, 1993, p. 1481–1484.

[HiWa95] H. Hild and A. Waibel. Integrating spelling into spoken dialogue recognition. In *Proceedings of the EUROSPEECH Conference*, Volume 95, 1995, p. 1977–1979.

[HiWa96] H. Hild and A. Waibel. Recognition of spelled names over the telephone. In *Proceedings of the 4th International Conference on Spoken Language Processing, ICSLP*, Volume 1. IEEE, 1996, p. 346–349.

[HTBRT06] D. Hakkani-Tür, F. Béchet, G. Riccardi and G. Tur. Beyond ASR 1-best: Using word confusion networks in spoken language understanding. *Computer Speech and Language* 20(4), 2006, p. 495–514.

[ICA01] *Aeronautical Telecommunications Volume II*, p. 58. International Civil Aviation Organization.
http://www.skybrary.aero/bookshelf/books/2279.pdf
(accessed 2014-03-06), 2001.

[Jeli05] F. Jelinek. Some of my Best Friends are Linguists. *Language Resources and Evaluation* 1(39), 2005, p. 25–34.

[JLMG93] D. Jouvet, A. Lainé, J. Monné and C. Gagnoulet. Speaker-independent spelling recognition over the telephone. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 2. IEEE, 1993, p. 235–238.

[JoGG07] M. Johnson, T. L. Griffiths and S. Goldwater. Bayesian Inference for PCFGs via Markov Chain Monte Carlo. In *Proceedings of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, HLT-NAACL*, 2007, p. 139–146.

[JVFM95] J.-C. Junqua, S. Valente, D. Fohr and J.-F. Mari. An N-best strategy, dynamic grammars and selectively trained neural networks for real-time recognition of continuously spelled names over the telephone. In *Proceedings of the IEEE International Conference on*

Acoustics, Speech, and Signal Processing, ICASSP, Volume 1. IEEE, May 1995, p. 852–855.

[Kelk07] B. Kelk. Phonetic Alphabets, 2007. http://www.w2aee.columbia.edu/phonetic.html (accessed 2014-03-06).

[KHBCB+07a] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin and E. Herbst. Moses – Confusion Networks, 2007. http://www.statmt.org/moses/?n=Moses.ConfusionNetworks (accessed 2014-02-10).

[KHBCB+07b] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin and E. Herbst. Moses – Word Lattices, 2007. http://www.statmt.org/moses/?n=Moses.WordLattices (accessed 2014-02-10).

[KINS+12] G. Kurata, N. Itoh, M. Nishimura, A. Sethy and B. Ramabhadran. Leveraging word confusion networks for named entity modeling and detection from conversational telephone speech. *Speech Communication* 54(3), 2012, p. 491–502.

[KSSW98] F. Kubala, R. Schwartz, R. Stone and R. Weischedel. Named entity extraction from speech. In *Proceedings of DARPA Broadcast News Transcription and Understanding Workshop*, 1998, p. 287–292.

[LaYo90] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language* 4(1), 1990, p. 35–56.

[LeCo03] J. R. Lewis and P. M. Commarford. Developing a Voice-spelling alphabet for PDAs. *IBM Systems Journal* 42(4), 2003, p. 624–638.

[Lee96] L. Lee. Learning of context-free languages: A survey of the literature. *Technical Report TR-12-96, Harvard University*, 1996.

[Leve66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10(8), 1966, p. 707–710. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[LoSp94] P. C. Loizou and A. Spanias. Context-dependent modeling in alphabet recognition. In *Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS*, Volume 2. IEEE, 1994, p. 189–192.

[LoSp96] P. C. Loizou and A. S. Spanias. High-performance alphabet recognition. *IEEE Transactions on Speech and Audio Processing* 4(6), 1996, p. 430–445.

[LWLF+97] A. Lavie, A. Waibel, L. Levin, M. Finke, D. Gates, M. Gavalda, T. Zeppenfeld and P. Zhan. Janus-III: speech-to-speech translation in multiple languages. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 1. IEEE, Apr 1997, p. 99–102.

[MaBS00] L. Mangu, E. Brill and A. Stolcke. Finding consensus in speech recognition: word error minimization and other applications of confusion networks. *Computer Speech and Language* 14(4), 2000, p. 373–400.

[Maha36] P. C. Mahalanobis. On the generalised distance in statistics. In *Proceedings of the National Institute of Sciences of India 2*, Nr. 1, 1936.

[MaSc94] M. Marx and C. Schmandt. Putting People First: Specifying Proper Names in Speech Interfaces. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, New York, NY, USA, 1994. ACM, p. 29–37.

[MeHi97] M. Meyer and H. Hild. Recognition of spoken and spelled proper names. In *Proceedings of the EUROSPEECH Conference*, 1997.

[MiSe99] C. D. Mitchell and A. R. Setlur. Improved spelling recognition using a tree-based fast lexical match. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 2. IEEE, 1999, p. 597–600.

[Moor04] R. K. Moore. Modeling data entry rates for ASR and alternative input methods. In *Proceedings of the INTERSPEECH Conference*, 2004.

[MSAV+11] J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, T. G. B. Team, J. P. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak and E. L. Aiden. Quantitative Analysis of Culture Using Millions of Digitized Books. *Science* 331(6014), 2011, p. 176–182.

[OeNe93] M. Oerder and H. Ney. Word graphs: An efficient interface between continuous-speech recognition and language understanding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 2. IEEE, 1993, p. 119–122.

[OgGo05] J. Ogata and M. Goto. Speech Repair: Quick Error Correction just by using Selection Operation for Speech Input Interfaces. In *Proceedings of the INTERSPEECH Conference*, 2005, p. 133–136.

[Oxf] Oxford Dictionaries. Definition of spelling in English. http://www.oxforddictionaries.com/definition/english/spelling (accessed 2014-03-08).

[Powe] R. Powers. Phonetic Alphabet. http://usmilitary.about.com/od/theorderlyroom/a/alphabet.htm (accessed 2014-03-06).

[Prof89] M. Proffitt. *The Oxford English Dictionary*. Oxford University Press. 1989.

[RaJu86] L. Rabiner and B.-H. Juang. An introduction to hidden Markov models. *ASSP Magazine, IEEE* 3(1), Jan 1986, p. 4–16.

[RaWi87] L. Rabiner and J. Wilpon. Some performance benchmarks for isolated work speech recognition systems. *Computer Speech & Language* 2(3), 1987, p. 343–357.

[RoRM97] F. Rodrigues, R. Rodrigues and C. Martins. An isolated letter recognizer for proper name identification over the telephone. In *Proceedings of 9th Portuguese Conference on Pattern Recognition*, 1997.

[SaSp04] M. Saraclar and R. Sproat. Lattice-based search for spoken utterance retrieval. *Urbana* Volume 51, 2004, p. 61801.

[ScRK00] H. Schramm, B. Rueber and A. Kellner. Strategies for name recognition in automatic directory assistance systems. *Speech Communication* 31(4), 2000, p. 329–338.

[SMFW01] H. Soltau, F. Metze, C. Fugen and A. Waibel. A one-pass decoder based on polymorphic linguistic context assignment. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU*, 2001, p. 214–217.

[Stol02] A. Stolcke. SRILM – An extensible language modeling toolkit. In *Proceedings of the 7th International Conference on Spoken Language Processing, ICSLP*, November 2002, p. 257–286.

[Suhm97] B. Suhm. Empirical evaluation of interactive multimodal error correction. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU*. IEEE, 1997, p. 583–590.

[SuMW99] B. Suhm, B. Myers and A. Waibel. Model-based and empirical evaluation of multimodal interactive error correction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1999, p. 584–591.

[ThRK00] F. Thiele, B. Rueber and D. Klakow. Long range language models for free spelling recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, Volume 3. IEEE, 2000, p. 1715–1718.

[TWGR+02] G. Tür, J. H. Wright, A. L. Gorin, G. Riccardi and D. Z. Hakkani-Tür. Improving spoken language understanding using word confusion networks. In *Proceedings of the INTERSPEECH Conference*, 2002.

[USF] US Financial Alphabet. http://www.thephoneticalphabet.com/financial-phonetic-alphabet.html (accessed 2014-03-06).

[YKHL+98] K. Younis, M. Karim, R. Hardie, J. Loomis, S. Rogers and M. DeSimio. Cluster merging based on weighted mahalanobis distance with application in digital mammograph. In *Proceedings of the IEEE 1998 National Conference on Aerospace and Electronics Conference, NAECON*, Jul 1998, p. 525–530.

[ZhHa05] R. Zhou and E. A. Hansen. Beam-Stack Search: Integrating Backtracking with Beam Search, 2005.