



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Interpreting voice commands in radar systems

Recognizing speech of multiple languages with limited resources
in an offline environment

Master's thesis in Computer Science

Johan Andersson
Oskar Dahlberg

MASTER'S THESIS 2017

Interpreting voice commands in radar systems

Recognizing speech of multiple languages with limited resources in
an offline environment

Johan Andersson
Oskar Dahlberg



Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Interpreting voice commands in radar systems
Recognizing speech of multiple languages with limited resources in an offline environment
Johan Andersson
Oskar Dahlberg

© Johan Andersson, 2017.

© Oskar Dahlberg, 2017.

Supervisor: Aarne Ranta, Chalmers, Department of Computer Science and Engineering

Supervisor: Sven Nilsson, SAAB AB

Examiner: Andreas Abel, Chalmers, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Computer Science
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2017

Automatic Speech Recognition in Radar Systems
Recognizing speech with limited resources in an offline environment
Johan Andersson
Oskar Dahlberg
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This master's thesis is done in collaboration with SAAB and outlines an architecture for developing a voice command system with focus on supporting multiple languages, ease of introducing new domain knowledge, resource efficient and running in an offline environment. The solution and the decisions made along the process as well as their advantages and drawbacks are explained in this report. As basis for the project we have worked with CMU Sphinx and Grammatical Framework to aid us in the process. Furthermore, the US-English community language model available from CMU Sphinx has been adapted to better understand the Swedish accent with the help of a tool provided by CMU Sphinx. The result of the adaptation as well as recommendation for future development is also included in this thesis.

Keywords: Speech Recognition, Sphinx, Grammatical Framework, Hidden Markov model, Syntax, Semantics.

Acknowledgements

We would like to thank our company partner SAAB AB for giving us the opportunity to work with such an interesting domain and problem in a meaningful environment. Furthermore we would like to thank the following people who have helped us carry out this Master's thesis.

Sven Nilsson, for supervising us at SAAB providing helpful feedback and guidance during the work and always keeping up a good spirit.

Aarne Ranta, for providing insights from his vast knowledge in the area and fast response time when questions emerged during the project.

Christian Fransson, who helped with understanding the domain and providing relevant commands as well as providing feedback on our design choices.

Martina Dinefjäll, for providing answers to any general questions about SAAB and helped out with pointing us to the right resources.

Ulf Dahlgren, who helped us with the integration in SAAB's own systems.

Carolina Fred, for helping us with grammar and word choice for the Spanish lexicon of our prototype.

We would also like to mention the following people who took time out of their daily work in order to gather training data for the project: Liza Axenskär, Lina Björnheden, Jeanette Davidsson, Andreas Eklund, Kent Falk, Christian Fransson, Martin Helmersson, Anders Karlsson, Kenneth Karlsson, Thomas Lang, Jakob Lundberg, Sven Nilsson, Esbjörn Rundberg, Elin Rönnow, Carl Schoster among others.

Johan Andersson, Gothenburg, June 2017
Oskar Dahlberg, Gothenburg, June 2017

Contents

1	Introduction	1
1.1	Aim	1
1.2	Delimitations	2
1.3	Related work	2
2	Theory & Method	5
2.1	Speech recognition	5
2.1.1	Preprocessing	5
2.1.2	Recognition problem	6
2.1.3	Language modeling	6
2.1.4	Acoustic modeling	7
2.1.5	Hidden Markov model	7
2.1.6	Recognition using HMMs	9
2.2	Parsing	12
2.2.1	Syntax	12
2.2.2	Semantics	12
2.2.3	Grammar	13
2.2.4	Available solutions	14
3	System	15
3.1	Overview	15
3.2	Speech-to-text	16
3.2.1	Sphinx	17
3.3	Grammatical Framework	18
3.3.1	Abstract syntax	20
3.3.2	Concrete syntax	21
3.3.3	Putting it all together	22
3.3.4	Usage of GF in our prototype	23
3.4	Interpreter	24
3.5	Integration	26
3.6	Summary of pipeline	26
3.7	Multilinguality	28
3.8	Model adaptation	28
4	Evaluation	29
4.1	Criteria	29

4.2	Experimental setup	30
4.2.1	Training step	30
4.2.2	Evaluation step	31
4.2.3	Software used	31
4.2.4	Hardware	32
4.3	Results	32
4.3.1	Implementing a new language	32
4.3.2	Runtime resources	33
4.3.3	Experimental results	33
4.3.4	Usability	34
5	Discussion	37
5.1	Discussion	37
5.1.1	Grammatical Framework	37
5.1.2	Experiment discussion	37
5.1.3	Precision	38
5.2	Critical system safety	38
5.3	Future work	39

1

Introduction

The introduction of speech control in everyday devices has enabled systems to be interacted with in a new way and has altered the expectations on systems in professional setting as well. Many speech recognition solutions such as Google Cloud Speech API [1], Alexa Voice Service [2], among others' require an online connection in order to interpret speech in a time efficient way.

An online solution is not always practical, for example due to poor internet connection or the security risk involved with connecting a device to the internet. In such cases an offline solution is required. Both of these reasons are true for the system we are targeting with our project and thus, our solution must work in an offline environment.

Another requirement, which is posed from our company partner SAAB (in the future also referred to as "the company"), is that the solution should be easy to re-adapt to spoken languages not previously incorporated. For example, if it can handle English, adding Swedish should be easy from a developer's perspective, while still yielding good results. This adaptability and having to run in an offline environment poses some interesting technical challenges each on their own.

1.1 Aim

The aim is to develop and evaluate a prototype application that can transform sounds into commands. This prototype is limited to 2 spoken languages, run in an offline environment and produce results in real time. The results of the prototype must be evaluated to measure its performance. Furthermore, it should not be overly cumbersome in terms of development time to introduce new domain knowledge or new spoken languages into the application. Parameters we have used for evaluation of the prototype consist of:

- Multilinguality, in our case how much work is required for adding a new language.
- Sentence/command error rate.
- Training/instructions required for the user to operate the system.
- Resource requirements in terms of computational resources.

The end prototype should be in a state where further development could incorporate it into a fully-fledged assistant application able to aid with domain-specific tasks.

1.2 Delimitations

To prevent the focus of the thesis from being too large, the scope is limited to the development of a prototype that works well with a limited feature set instead of a larger system that has poor performance. For instance, this limited feature set does not cover background noise reduction or speaker identification.

Time is also a limiting factor which means that third-party libraries have to be used in order to complete the prototype within the timeframe. It is not feasible to create a new speech-to-text system and at the same time gather data in order to train new models. Rather, we use existing models in order to save time and being able to focus on other parts of the system.

The prototype implementation covers only two spoken languages. This is due to the limited timeframe in which the prototype has to be developed, but also the number of languages spoken by the authors. Since the authors are only fluent in two languages; Swedish and English, English is the main focus for the prototype with adding another language as proof of concept. There is a larger focus on making it simple for the system to include a new spoken language rather than implementing as many languages as possible.

1.3 Related work

A Ph.D. student has done a similar project in 2005 where a multilingual device was developed to help individuals navigate Gothenburg [3]. It could interpret several languages, including English, Swedish, and Spanish. The project consisted of a map representing the tram network, it was used by either saying the names of the stations out loud or by selecting the stations on the map. It also gave responses in the form of speech. This approach also used Grammatical Framework (see section 3.3) as a tool to help with multiple languages, but had other solutions for the remainder of the system.

The main difference from our project is the domain and the requirements that come with it. While the project done in 2005 worked as a tool for pedestrians to take the tram, our project will work as a tool for a professional operator to use as an addition to existing control system. There are higher demands on response time and precision. The previous system was limited to very few commands of the same structure while our project is focused on a broader sense with considerable more command variations.

There has also been a project constructing an automatic speech recognizer on the International Space Station (2005). The authors implemented both a statistics-based model (see section 2.1.3 on language modeling) and a grammar-based model (see section 2.2.3 on grammar). It is a command-based system which uses phrases like “set alarm for five minutes from now” and “please speak up” [4]. This system only understands English, but it extracts the semantics in a way similar to Grammatical Framework. The authors also concludes that the word error rate is lower in a system which uses a grammar-based approach, than that of a system which uses

a statistics-based approach. The main difference to our project in this case is that of multilinguality, understanding more than one language.

2

Theory & Method

In order to have computers recognize and understand human speech it is beneficial if the input is converted into a format that the computer can easily process, for example a text string or some other data structure. This chapter will briefly explain theory and available methods for doing natural language processing from speech to help the reader follow the subsequent chapters.

2.1 Speech recognition

In Figure 2.1 we can see an overview of a speech recognizer pipeline taking audio signal input and outputting a transcription. In the following section we will explain the theory behind this process and what different parts are necessary to perform recognition.

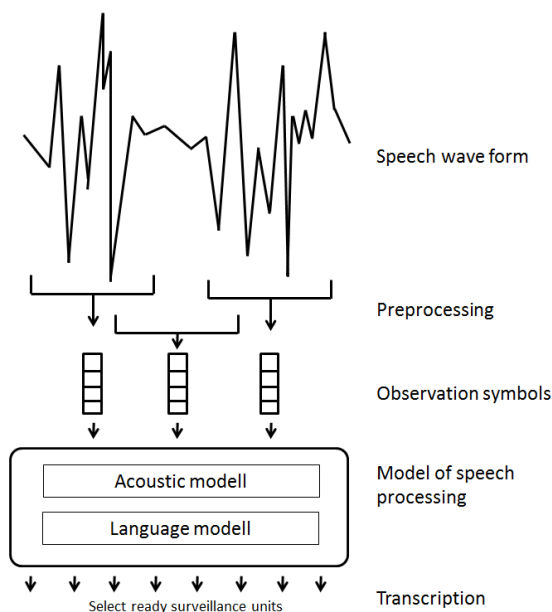


Figure 2.1: Overview of recognizer process. Adapted from Ir P Wiggers and LJM Rothkrantz [5]

2.1.1 Preprocessing

Speech is a continuous signal with infinite possible variation depending on a lot of factors such as speaker mood, environment, and background noise among others.

However, not all information present in this variation is needed to perform speech recognition; it is therefore advantageous to reduce the input space to simplify the complexity of the model used to perform recognition. The first step of speech recognition is thus a preprocessing step to reduce the input and highlight important features.

During the preprocessing step the continuous input is split into a discrete set of observations. In our case these are feature vectors produced by a variant of Mel-frequency cepstral analysis [5]. The process of Mel-frequency cepstral analysis takes audio signal as input and performs a series of transforms and filtering operations and gives a vector as output. First the input audio is split in pieces of, for example, 25ms of snippets sampled every 10ms, giving overlapping snippets. On these snippets Mel-frequency cepstral analysis is performed resulting in speech vectors.

Humans are much better at determining changes in pitch in lower frequencies than higher [6]. Mel-frequency cepstral analysis utilizes this knowledge by filtering away redundant parts humans would not perceive. The resulting vector captures the details of human speech while removing noise and other needless parts of the input data. This results in a compact format keeping the important bits.

2.1.2 Recognition problem

After the preprocessing step the input has been divided into a sequence of discrete values. Those values, or observations are then used to determine one or many hypotheses on what was said. The recognition problem can then be stated as follows: We have a sequence of observations $O = o_1 o_2 \dots o_n$, given these observations and a set of all possible sequences of words spoken, S , we want to determine the most likely sequence of words W . This can be done by solving the equation 2.1, where w is an element of S .

$$W = \underset{w}{\operatorname{argmax}} P(w|O) \quad (2.1)$$

This problem however is not easily solvable since possible observation sequences, even with the preprocessing step, are very large and S is infinite. Bayes' theorem allows us to reformulate this equation to the equation 2.2. This leads us to the two main components of a recognizer, the language model $P(w)$ (see section 2.1.3), and the acoustic model $P(O|w)$ (see section 2.1.4). The term $P(O)$ can be ignored since it is simply a scaling factor. If we can solve this new problem, then we can output a hypothesis for a given observation. To output a set of hypotheses is only a matter of changing the argmax to not return a single instance but a list of instances ordered by likelihood where the first item is most likely.

$$P(w|O) = \frac{P(w)P(O|w)}{P(O)} \quad (2.2)$$

2.1.3 Language modeling

A language model is a probability distribution over a sequence of words, $P(w)$. Given a sequence of words, a language model can answer how probable that sequence is.

When it comes to speech recognition, language models can be used in order to determine which word was said if there are many alternatives from the phonetic analysis. For example, for the phrase “I will dye my hair” the language model will assign a higher probability for the word “DYE” than the word “DIE” which have very similar pronunciations.

Given the amount of variations to write a particular text and the breath of vocabulary in many languages, the probability of observing a particular sequence decreases as the sequence length increases. For example, this particular paragraph is probably unique in its particular choice of word and order. Since longer sequences will fast approach a probability of zero, it is common to limit the probability of any given word to be dependent on n previous words. This is known as an n -gram model [7]. Commonly n is between 1 and 3 to reduce model size since the amount of data needed to train the model grows very fast as n increases.

2.1.4 Acoustic modeling

An acoustic model answers the question of which observations O are produced when a particular sequence of words w is uttered. Given the nature of speech and the unbounded length of such a sequence, in order to solve this problem we need a statistical model where answers can be computed on the fly [5]. In other words, an acoustic model can calculate the probability $P(O|w)$ for any O and w . Commonly, most acoustic models are instances of a Hidden Markov model (HMM) [8], although it is possible to use other models such as artificial neural networks.

2.1.5 Hidden Markov model

A Hidden Markov model is a variant of Markov model [9] where the observation sequence and state sequence are decoupled [5]. Every state has a probability distribution which determines how likely a particular observation is in that state. Given a particular observation it is not possible to determine which state generated it, the state sequence who generated a particular observation sequence is unknown from looking at the observations, they are hidden. A Hidden Markov model can be defined seen in list 2.1. A model with all parameters defined can thus be written in short form as $\lambda = (A, B, \pi)$, where A , B , and π have some appropriate values.

Additionally when it comes to speech recognition and the usage of HMMs there are some things to consider. Speech is modeled as a set of observations ordered in a sequence. To capture this order when using HMMs to recognize speech the HMMs used are of the type left-right models, the states are presented as a chain and the probability of transitioning backwards in the chain is set to 0 an example of a left-right HMM can be seen in Figure 2.2.

How are HMMs used to recognize speech then? If we assume we have a set of models already trained to recognize different words in the form of HMMs. For a particular observation sequence O we want to determine which model is most likely to generate that sequence, we want to determine $\operatorname{argmax}_{\lambda} P(O|\lambda)$. Once this is done we can see which model, λ , was returned and lookup which word that particular model corresponds to.

- Number of distinct observation symbols M .
- An output alphabet $V = v_1, v_2, \dots, v_M$.
- Number of states N
- State space $Q = 1, 2, \dots, N$
- Probability distribution of transitions between states $A = a_{ij}$ where $a_{ij} = P(q_{t+1} = j | q_t = i)$ $1 \leq i, j \leq N$
- Observation symbol probability distribution $B = b_j(k)$ in which $b_j(k) = P(o_t = v_k | q_t = j)$ $1 \leq k \leq M, 1 \leq j \leq N$
- Initial state distribution $\pi = \pi_i$ where $\pi_i = P(q_1 = i)$ $1 \leq i \leq N$

List 2.1: Definition of Hidden Markov model [5]

To solve this problem the calculation in equation 2.3, where q is a state sequence, can be evaluated. However, this calculation quickly becomes unfeasible to do in real time as the complexity grows with the number of states and exponentially with the length of the observation sequence, resulting in a complexity of $O(TN^T)$ where T is the length of the observation sequence. For a more detailed explanation on this equation and the complexity of it see Wiggers and Rothkrantz work [5].

$$P(O|\lambda) = \sum_q P(q|\pi) \prod_{t=1}^T P(o_t|q_t, \lambda) \quad (2.3)$$

There exist a method to perform this computation in a feasible way though, the Forward algorithm [9]. In the Forward algorithm a dynamic programming approach is utilized to calculate the values for each time step in parallel. Its efficiency comes from the fact that many sequences share common subsequences and performing the calculations in this manner prevents recalculating many subsequences along the way. The complexity of the Forward algorithm is $O(N^2T)$ which is considerably better than the case before and can be used in practice.

Given HMMs for different words it is possible to recognize them. But how can such models be created? We have the shorthand for an HMM definition as $\lambda = (A, B, \pi)$. The question is now how we estimate A , B and π to create a model reliably recognizing the word we want. The parameters we are interested in are the transition probabilities between states, how likely each observation is in each state as well as the initial distribution of states. This cannot be done without knowing what we want to recognize, thus we require a set of observation data D in order to estimate the parameters. Given that we have data covering what we want to recognize we want to create a model as seen in equation 2.4.

$$\lambda_{trained} = \operatorname{argmax}_{\lambda} P(D|\lambda) \quad (2.4)$$

This is not analytically solvable but there exists an algorithm solving it iteratively; Baum-Welch algorithm or Forward-Backward algorithm [10]. The algorithm is a version of Expectation Maximization (EM) algorithm adapted to fit the case of HMM. The idea is to start with some initial estimation of the parameters, compute expectations on transitions and output of observations and then update the

estimations of the parameters. This process repeats until convergence. During the process the estimation step utilizes the observation data. It is a type of hill-climbing algorithm, which means it does have the risk of getting stuck in a local maximum.

2.1.6 Recognition using HMMs

In terms of Speech Recognition when it comes to using HMMs there are some things to consider. As previously mentioned, states are usually placed in sequence without any transitions backwards in this sequence. Another design decision is what an HMM should model. Previously, our example for the recognition mentioned several HMMs where each of them represented a word on their own. This is not necessary, for example an HMM in speech recognition can model a phrase, a word or a sub-word unit.

Those different kinds come with advantages and disadvantages. While an approach where HMMs represent words can be very useful as it can capture coarticulation effects it requires that all words that we want to recognize are part of the training data, preferably in many samples [5]. To reduce the required amount of training data it is common to use sub-word units. Different words can contain the same sub-word unit, which means that the HMM representing a specific sub-word unit can receive training data from different words. Training data can be shared and it is possible to capture words that are not present in the training data if all their sub-word parts are present.

For example, if we consider the words “select” and “celebrity” both share some parts of pronunciations, written in a type of phonetic text they can be written out: “/S/ /AH/ /L/ /EH/ /K/ /T/” and “/S/ /AH/ /L/ /EH/ /B/ /R/ /IH/ /T/ /IY/” [11] respectively with each sub-word unit marked out. If we train a model where both those words are present using sub-word units HMMs, then this model could be able to recognize the word “celery” which corresponds to “/S/ /EH/ /L/ /ER/ /IY/” since most sub-words units are in the other words present in the training data, given that there exist words with the sub-word unit “/ER/” (since it is missing in the other two words) in them within the training data.

A typical use of sub-word models are phones [12]. A phone is a unit of sound. A pair of consecutive phones is called a diphone. Along the same line, three consecutive phones are called a triphone. Diphones and triphones are used to capture the properties of transition between sounds and their relation. Having a way to model such transitions and sounds themselves is necessary to capture the properties of human language. Since different languages use different sounds or at least different order of sounds there are different sets of phones, diphones and triphones for each language. Furthermore, a phone is generally considered to have 3 phases; an on-glide phase, the pure phase and, the off-glide phase [13]. The first and last phases consist of the phone with an overlap of proceeding and succeeding phones respectively, while the pure phase in between is the pure form of the phone.

How does the sub-word model of phones, the pre-processed signal in terms of feature vectors and HMMs fit together then? Initially, speech vectors were mapped to different phones using K-means clustering, likely a result of limited computing power and memory. Nowadays, feature vectors are used directly in the HMMs.

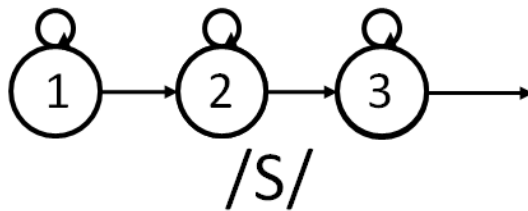


Figure 2.2: HMM representing the phone /S/

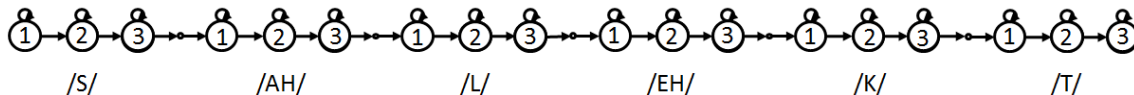


Figure 2.3: HMM representing the word “select”

This means that the observation symbol probability distributions are in the form of multivariate continuous probability density functions, $\mathcal{N}(\cdot)$, as seen in equation 2.5 where u_j is mean vector and U_j is the covariance matrix.

$$b_j(o_t) = \mathcal{N}(o_t, \mu_j, U_j) \quad (2.5)$$

Given the previous statement of phones having 3 phases an HMM representing a phone typically has at least 3 states, one for each phase. An example of such an HMM can be seen in figure 2.2. It is possible to connect HMMs to each other creating a new larger composite HMM. Connecting many HMMs representing different phones we can create an HMM representing a whole word, an example of this can be seen in figure 2.3 where a composite HMM for the word “select” is shown.

In a similar way we can connect words together. We mentioned previously that words are usually modeled in a way where a particular word depend on previous words. This can also be viewed as a particular word can be followed by a set of words with a certain probability for each of them. In a way this can be seen as a Markov model where each state is a word and the transitions captures the probability of a word being followed by another word. Using this fact we can create a composite HMM consisting of words where we replace the words by their phone HMMs to create a model for a whole phrase. An example of such an HMM can be seen in figure 2.4.

Note, that using such composite HMMs risk creating large models, which can become hard to search. To circumvent this problem it is common to prune away parts of the model along the way. For each time step, sequences which do not meet a certain threshold are terminated and not considered for later time steps.

We now have all the building blocks required to transform audio signal into a transcription of words through the use of HMMs.

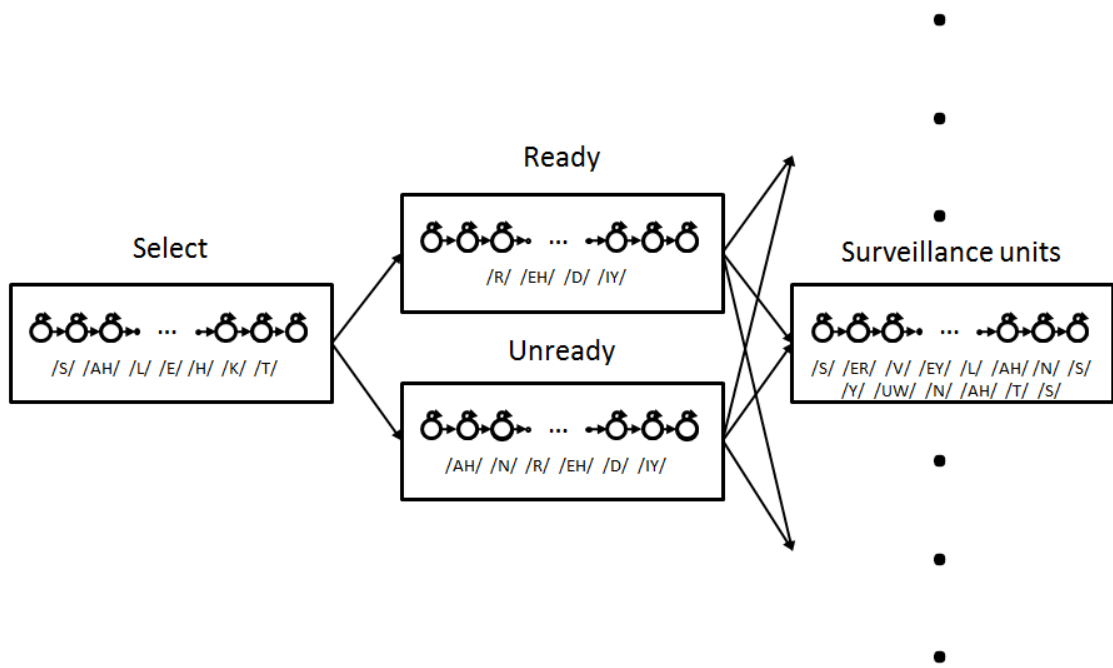


Figure 2.4: HMM representing the phrase “select ready/unready surveillance units”

2.2 Parsing

Through the use of an acoustic model and language model it is possible to convert raw audio signal into text. While a text string of natural language is an improvement it is not easily managed format of input to an application. Problems such as ambiguity remain among other concerns such as superfluous information. The string “Please, pass me the salt, kind sir” contains a lot of unnecessary words that are not needed to represent the command given. Doing some processing of previous mentioned sentence could put it in a more structured format such as: {action: move, source: salt, target: speaker, worker: person}. Performing such an extraction and mapping of information is no trivial task however [14].

In order to solve this information extraction and mapping issue there are many approaches available, it could be possible to use a statistical method or machine learning (ML) method with the use of Artificial Neural Networks (NN) for example. In both those cases there is the requirement of previous data and the end result is always probabilistic in nature in some way. When designing a system such as ours where precision is a key element, this approach is undesirable due to the risk of errors. While NNs may perform well it is hard to reason about why they perform well and in what situations they may fail. A more transparent method and hand engineered solution is desirable to easier reason about and explain system performance and reliability.

Our chosen approach is to use a parser. Commonly parsers are built using a grammar-approach [15] where the developer specifies a syntax in the form of a grammar that the parser then uses to convert raw text into a parse. The representation of a parse is often in the form of a tree as it can then easily represent the order and relation of different tokens [15].

2.2.1 Syntax

The definition of syntax used in this thesis is the one concerning natural language as given by the Oxford dictionary: “The arrangement of words and phrases to create well-formed sentences in a language.” [16] To elaborate, syntax describes how a language is built, in what order do the words come, how can a word with the same meaning be said in different ways and when should it be used, for instance plural or masculine/feminine. Syntax is a way to describe these rules e.g., should the adjective come before the subject as in English, or the other way around as in Spanish? Each language has its own set of rules when words should be used which can lead to difficulties when translating from one language to another [17], as well as when trying to develop a software system able to process human language. For each language that should be processed by the system that particular language syntactic rules must be incorporated in the system for it to function properly.

2.2.2 Semantics

While syntax describes *how* something is written, semantics describes *what* is being written, the meaning of the text [18]. Understanding semantics is the part of nat-

ural language understanding that is most difficult for computers. For example, the headline “Criminals get nine months in violin case” [19], is easier for a human to understand that the headline is likely about a court case and not a physical violin container, but a computer without more than dictionary knowledge would not be able to tell.

2.2.3 Grammar

In terms of parsing and text processing, a grammar is a way to express syntactic rules and capture legal sentences in a language. See Figure 2.5 for a simple grammar in JSpeech Grammar Format covering phrases like “please open a window thanks” and “could you close the file” among others. The grammar has a starting point with the rule for a basic command seen in Figure 2.6 which consist of a polite start, the command followed by a polite end. Then follows rules that define a command as an action and an object and so on.

```
public <basicCmd> = <startPolite> <command> <endPolite>;

<command> = <action> <object>;
<action> = open | close | delete | move;
<object> = [the | a] (window | file | menu);

<startPolite> = please | kindly | could you ;
<endPolite> = [ please | thanks | thank you ];
```

Figure 2.5: Example of JSpeech Grammar Format [20]

```
<basicCmd> = <startPolite> <command> <endPolite>;
```

Figure 2.6: Example of rule in JSpeech Grammar Format

The benefit of using a grammar-approach in our case is twofold. First, there is very little development cost of modifying or expanding an existing grammar. While it may not be trivial to do in all cases it can be as easy as adding a single line in a text file to add a completely new phrase. There is no need to retrain the system, as would be in a statistical ML-approach, or do other modifications.

Secondly, it is possible to improve the precision of speech-to-text tools through the use of grammar [4]. Instead of deciding through probability between possible words in the language model it can simply disregard words not present in the grammar, the grammar is the language model. The grammar may still be ambiguous and the output may be many possible parses with a probability score assigned to each of them.

The downside is that the system only recognizes the words specified by the grammar. If the user says something similar, the system cannot determine what the user meant, or it tries to fit the sentence into the given grammar, which can

result in false positives. For example, if the grammar contains the word “remove” and the user says the similar word “approve”, which the grammar does not include, the system risk matching the input as “remove” and thus misinterprets the user.

2.2.4 Available solutions

There exist a number of different approaches to do parsing. One way is to have a regular language described by a set of regular expressions and have a regular expression tool create a parser for the particular language. Another way, common to programming languages, is to have their syntax specified in a deterministic context-free grammar (DCFG). The reason for using DCFG is that it is possible to write parsers for a DCFG that can parse input in linear time [21]. However, using DCFG comes with limitations, for instance it does not allow ambiguity which makes it unsuitable for parsing natural language where ambiguity often is present. There is no need to limit the model to a DCFG if the strings are short (phrases, compared to large code bases), the gain from linear time does not outweigh the gain from a more refined parsing approach allowing ambiguity.

An alternative to classic deterministic grammar based parsing is to use statistical approaches with help of machine learning algorithms. For such approaches, data is first gathered and labelled manually as a basis for training the system. Such systems require large sets of data in order to work efficiently and risk suffering from common problems such as overfitting [22]. An alternative approach we have explored is to use a grammar approach that allows ambiguity and allows separation of semantics from syntax, for this purpose we have explored Grammatical Framework, which is further explained in section 3.3.

3

System

There are many ways to approach the challenge of building an Automatic Speech Recognition (ASR) system, a good outline is to start with the goal of the system and adjust approach thereafter since many available technologies are suited better or worse depending on situation. This chapter will discuss the different techniques used and which ones have been disregarded.

3.1 Overview

The problem at hand, developing an ASR system interpreting natural language into specific computer understandable commands, is split into many modules and sub-problems.

As can be seen in Figure 3.1, the prototype is divided into four parts that send information from one step to the next. The first step takes speech and turns it into a format that is easier to analyze. In this case the speech-to-text module produces a text for the next module to analyze. The second part is the parser which transforms given input natural language text into a format that is easier to abstract information from, a parse tree. The third part then takes a parse tree as input and interprets if this particular input is valid in the current context or helps disambiguate if several interpretations are possible and outputs valid interpretations. The fourth and last part, the application layer, then receives an interpreted command and executes actions corresponding to that command.

Since there are multiple steps during the process, it is easy to follow the transformation and see where an error occurred if something went wrong. Furthermore, it is easy to replace any part of the pipeline if a better alternative is found. However, there is the risk of reducing performance by keeping the different parts separated. There is the possibility that having more communication and closer connection between the parts could help performance, perhaps a step used later in the pipeline could be involved earlier for a better result? We deemed the advantages to be limited at best and considered the architecture described in Figure 3.1 to be better suited for our problem.

Such a modular approach does not necessarily prevent different parts providing information to the others, for example the parser module model is used for the speech-to-text module as well in a different format. Furthermore, it would be possible to have a probabilistic approach where each step had several output paired with confidence scores. For example, the speech-to-text module could output several transcriptions with the same confidence while later steps could refine this by

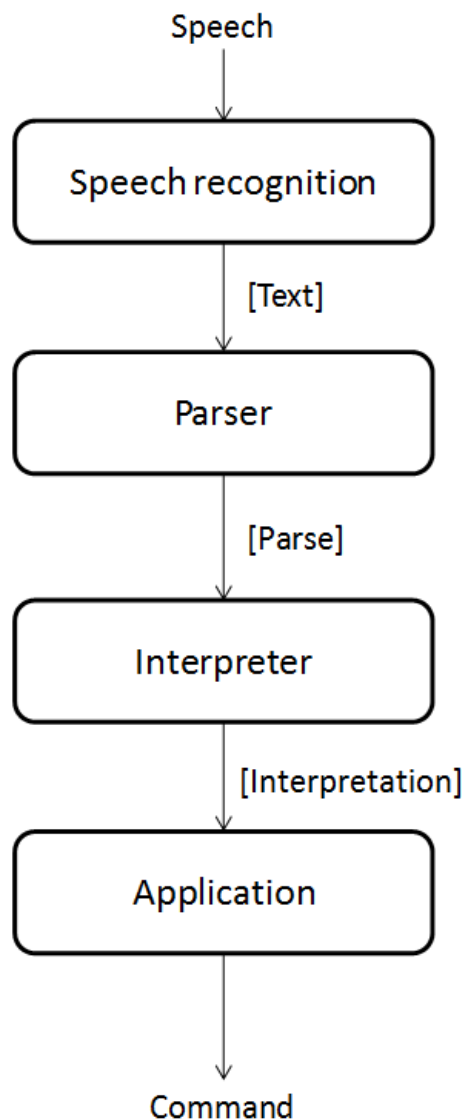


Figure 3.1: Overview of prototype pipeline

adding or subtracting to these scores. Unfortunately, our speech-to-text software tool only outputs one hypothesis when using grammars which prevented us from doing a probabilistic approach.

3.2 Speech-to-text

The first part of our pipeline is the speech-to-text module; it is responsible for converting raw audio signal into text. During this section we explain our prototype setup and the techniques and libraries chosen. For more details on general theory how this process is done see the theory section 2.

There is a limited amount of tools to choose from when exploring ASR with the requirement of running in an offline environment in a multiplatform setting. After some research we filtered out three possible alternatives: HTK Speech Recognition

Toolkit (HTK) [23], Kaldi ASR (Kaldi) [24] and, CMUSphinx (Sphinx) [25]. They can all run offline and claim to be more or less resource effective.

HTK Speech Recognition Toolkit (HTK) and Kaldi ASR (Kaldi) were later disregarded for different reasons. HTK has a license that does not allow for commercial use or redistribution. Furthermore, HTK is implemented in C which is not compatible with the company partner’s requirements, since C is deemed an unsafe language. Kaldi is a toolkit developed as a research tool, mainly for experimenting with the latest technologies rather than being easy to use in applications. It is implemented in C++, which the authors have limited experience in using as well as it not being on par with the company partner’s requirements. Kaldi has indirect support for grammars but it would require some manual transformation between formats, from a grammar into finite state transducer [26].

Sphinx is a speech recognition toolkit focused on practical application development and not on research, designed specifically for low-resource platforms [27]. It seems well established within the software community, it has good documentation and getting hold of the developers to ask questions has been easy. It is written in Java which is one of the main languages used by our company partner; this means that any future development and integration of our prototype should be easy. It supports using grammars in the JSpeech Grammar Format in the recognizer out of the box which helps with increasing precision. However, while using grammars it does not expose any support for getting confidence scores in any meaningful way, to our knowledge. Sphinx also seems to report half-recognized constructs in the grammar rather than reporting an error if the user does not complete a sentence. In the end Sphinx was chosen as it fulfilled our requirements and best fit our purpose compared to HTK and Kaldi.

3.2.1 Sphinx

For our prototype we use Sphinx4, the Java version of Sphinx’s tool stack. We have used their community created acoustic models for US-English and Spanish for our prototype as well as the dictionaries provided with them. Alongside Sphinx community models we use our own grammar exported from the Grammatical Framework. All this allows us to run Sphinx from within our Java application using their models and transforming audio signal from a recording device connected to a computer into Java strings. The recognizer runs on its own thread to prevent blocking other operations and reports its results to a worker queue taken care of by the rest of the application.

For example, the phrase “select ready surveillance units” uttered by the user will go through Sphinx preprocessing steps of converting the audio to feature vectors and removing background noise. Afterwards the community created acoustic model and our grammar language model work in tandem checking the input against the internal HMMs present, the first audio snippets are matched against the phones “/S/ /AH/ /L/ /EH/ /K/ /T/” which corresponds to “select”, without risk for matching against “/S/ /AH/ /L/ /EH/ /B/ /R/ /IH/ /T/ /IY/” which corresponds to “celebrity” and is not allowed by our grammar language model. When the whole

input is processed we receive a hypotheses from Sphinx: “select ready surveillance units” which is added to a queue for further processing in the pipeline.

While Sphinx in its current state does not expose confidence values for hypothesis when grammars are used, it is possible that such support could be added to the Sphinx code base. Since Sphinx is open source it would be possible to look into how the grammar speech-recognizer works and possibly add support for exposing its confidence values. How difficult or easy this would be is hard to tell since the limited time-frame and other priorities have not presented the opportunity to explore Sphinx code base in depth.

3.3 Grammatical Framework

The aim of the prototype is to support multiple languages, thus, adding a new language should not require a completely new system. If a generic system was adding support for more languages, by simply matching specific sentences to commands, the number of matches would be growing with a factor of $k \cdot n$, where k is the number of commands and n is the number of languages. Since each new command would have to, in its full form, be matched in each language that is supported. Such a growth in development complexity is undesired and thus a more general approach is needed to resolve this. It is possible to extract shared semantics of languages into a more abstract form, so sentences from different languages that share semantics can be transformed to the same abstract form. For instance, in English “turn the lights off” and in Swedish “släck lampan” has the same semantic meaning while their syntax and words differ. If we in the parser module can extract the semantic meaning and output it to the next part, the next part can then work independently of the input language. There is a tool that has aided us in this effort: Grammatical Framework (GF) [28].

GF allows a user to specify an abstract syntax that covers semantics you want to extract and then for each language you want to support you implement a concrete syntax following the abstract syntax’s specification. This allows developers to encode their domain in an abstract syntax and only port parts that are language dependent in the concrete syntax. GF can then parse any language having a concrete syntax into an abstract parse tree. To summarize, given any input language with a corresponding concrete syntax it can produce an abstract syntax tree which is independent of the input language. Thus the phrases “markera redo övervakningsenheter” and “selecciona unidad de vigilancia preparada” is parsed by GF into the same abstract parse tree, examples of parse trees can be seen in figures 3.2, 3.3 with their abstract text representation visible in figure 3.4. GF with its help library covers all major languages and helps with working out language specific syntax which allows developer not familiar with the linguistic properties of the language to use GF with its library to incorporate an unfamiliar language.

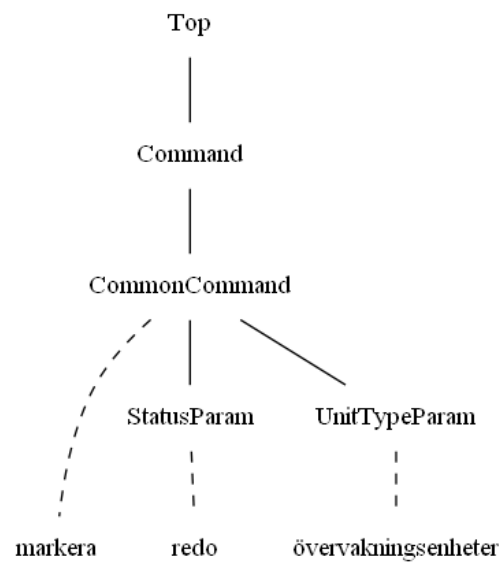


Figure 3.2: Parse tree for Swedish

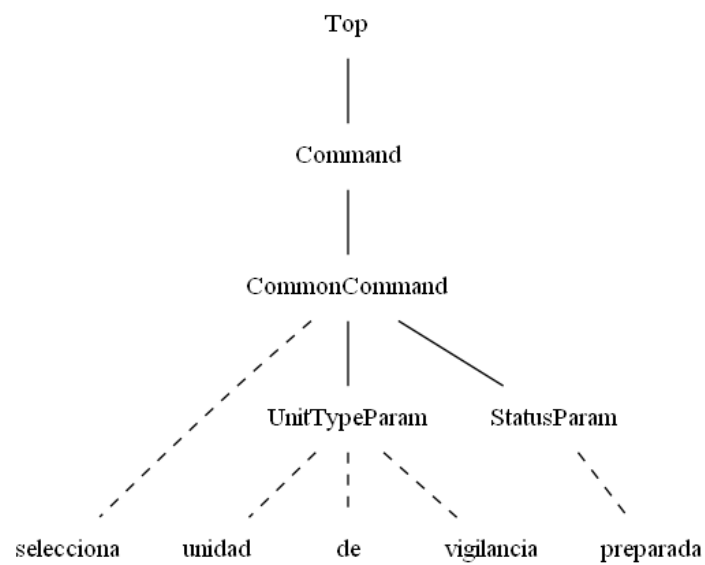


Figure 3.3: Parse tree for Spanish

TopCmd (CommonToCmd (SelectUnitsCmd Surveillance Ready))

Figure 3.4: Abstract Parse Tree from GF

```
-- Unifying constructs
TopCmd      : Cmd -> Top ;
CommonToCmd : CommonCmd -> Cmd ;
SensitiveToCmd : SensitiveCmd -> Cmd ;

-- Actual commands
GetDirectionCmd : PID -> CommonCmd ;
GetSpeedCmd    : PID -> CommonCmd ;
GetDistanceCmd : PID -> CommonCmd ;
GetClosestOfType : TargetParam -> CommonCmd ;
SelectUnitsCmd : UnitTypeParam -> StatusParam -> CommonCmd ;
ClearSelections : CommonCmd ;
SetModeCmd     : Mode -> SensitiveCmd ;
DisplayRangeCmd : ShowHideParam -> PID -> CommonCmd ;
DisplayInformationCmd : PID -> CommonCmd ;
```

Figure 3.5: Extract of abstract syntax

There are many advantages using GF. It allows us to simplify the process of extracting semantics from different languages into a shared language independent output of the parser module. GF can output its grammars in many formats, including JSpeech Grammar Format (JSGF) which our Speech-to-text module can then use as language model. Furthermore it allows language specific output given an abstract syntax tree which allows us to have generic output of our application be linearized into the user’s own language which can vary.

3.3.1 Abstract syntax

For our prototype we have created a small example grammar in GF within our domain, allowing the user to query for information and carry out actions in the prototype application. An extract from the abstract syntax can be seen in figure 3.5 covering some commands. Here we see different constructions for the different commands with varying number of variables required for them. It resulting in a “CommonCmd” or a “SensitiveCmd” which is unified to our “Top” category by the first three rules in the example. Since there are several types of commands, the system is able to recognize this depending on the label you give it; the system is then able to have different behaviors for different labels of the command. For instance, a “SensitiveCmd” might require that you accept the action before it executes it, since the instruction has a larger effect.

If we take a closer look at the command in figure 3.6 we can see that the “SelectUnitsCmd” takes two arguments, a “UnitTypeParam” and a “StatusParam” which are later in the grammar defined as seen in figure 3.7 with different constructors. This rule defines that in order to select units it requires the type and the status of those units in order to construct a meaningful command. This is all the information we need in the abstract syntax for our example sentence, “select ready surveillance units”.

```
SelectUnitsCmd : UnitTypeParam -> StatusParam -> CommonCmd ;
```

Figure 3.6: Abstract construct linked to “select ready surveillance units”-command

```
Surveillance : UnitTypeParam ;
Ready, Unready, Unarmed, Armed : StatusParam ;
```

Figure 3.7: Extract of abstract syntax

3.3.2 Concrete syntax

After the abstract syntax comes the concrete syntax filling the specification. We have used GF’s functor structure to clearly separate different parts and have a partial, or as in GF-terms, incomplete concrete syntax to define word types and general rules for the different rules of the abstract syntax. In figure 3.8 we can see the rule in the concrete syntax for our selection-command. First we have the name of the rule followed by the parameters named u and s ; this is followed by an implementation taking help from GF’s help library for grammatical rules.

The implementation consists of an ordered set of function applications. All commands are utterances and thus have the “mkUtt” construct wrapping them. For this particular case we want imperative form of the verb “select” which acts on a particular noun “ready surveillance units” which leads up to the usage of “mkImp”, “mk_V2” and “mkNP” where “Imp”, “V” and “N” is short for imperative, verb and noun respectively in the GF helper library api. Since the verb acts in regard to something we are looking for a two-place verb and noun in question have modifier in the terms of “ready” which leads up to the usage of “mk_V2” and “mkCN”. The last function used, “mkNP”, simply unifies nouns to noun-phrases. This results in an utterance in imperative form with verbs acting on a particular noun. The “Select_V” construct and constructs for status and unit arguments are the only things missing, all other functions are provided by the GF help library. The argument constructs can be seen in figure 3.9 and are simple delegation to the last part; the lexicon part of the syntax.

The lexicon part consists of rules which describe what constructs maps to what words in that particular language. This is the only language dependent part of our grammar and consists of as little glue as possible with only lookups and mostly simple unary functions as well as some mapping to the language paradigm of GF helper library to a particular language. This can all be seen in our extract in figure 3.10 from our lexicon implementation of English. The only difference between the

```
SelectUnitsCmd u s = mkUtt (mkImp (mk_V2 Select_V) (mkNP (mkCN s u))) ;
```

Figure 3.8: Concrete construct linked to “select ready surveillance units”-command

```

Surveillance = Surveillance_N ;
Ready = Ready_A ;
Unready = Unready_A ;

```

Figure 3.9: Concrete part delegating to lexicon part

```

Ready_A = mkA "ready" ;
Unready_A = mkA "unavailable" ;
Surveillance_N = mkN "surveillance units";

Display_V = mkV "display" | mkV "show" ;
Select_V = mkV "select" ;
Clear_V = mkV "clear" | mkV "remove" ;

-- Delegations to GF lib
mk_V2 = mkV2 ;
mk_V3 = mkV3 ;
mk_ID_N = mkN ;
mk_N2 = mkN2 ;

```

Figure 3.10: Lexicon part of concrete syntax for “select ready surveillance units”-command for English

English, Swedish and Spanish grammar is the lexicon part, in figures 3.11 and 3.12 the differences required for our example can be seen.

3.3.3 Putting it all together

The relation between the different parts are covered in part by inheritance syntax of GF as well as a functor instance which maps required interfaces to their respective instance. An example of such wiring can be seen in our Swedish grammar in figure 3.13, this maps the Syntax module required by the GF library functions to the Swedish one in the library as well as the lexicon for our grammar to Swedish lexicon we have created. When all these parts are in place it is possible to invoke GF with a parse instruction and give it the example sentence of “markera redo övervakningsenheter”, “select ready surveillance units” and “selecciona unidad de vigilancia preparada” and all of those inputs will produce the abstract tree “TopCmd (CommonToCmd (SelectUnitsCmd Surveillance Ready))” as a result.

```

Ready_A = mkA "redo" ;
Select_V = mkV "markera" ;
Surveillance_N = mkN "vervaknings" (mkN "enheter");

```

Figure 3.11: Minimal part of lexicon for “select ready surveillance units”-command for Swedish


```

Ready_A = mkA "preparado" ;
Select_V = mkV "seleccionar" ;
Surveillance_N = mkN "unidad de vigilancia";

```

Figure 3.12: Minimal part of lexicon for “select ready surveillance units”-command for Spanish

```

concrete FutureSwe of AbstractFuture = Future with
  (Syntax = SyntaxSwe),
  (LexFuture = LexFutureSwe);

```

Figure 3.13: Functor instance of Swedish grammar

While all of the above may seem like a lot of effort in order to parse a line of text there are benefits to using GF and doing it in this particular way. There is a clear distinction between what you seek in the domain, which is captured by the abstract syntax, and the language specific implementation, which is captured by the concrete syntax. The concrete syntax uses proper grammatical constructors with the help of all common language constructors without having to worry about the language specific syntax. Once the abstract grammar and the first part of the concrete grammar are in place, the matter of introducing a specific language is as simple as filling out a lexicon file, which is the second part of the concrete grammar, with proper word choices for that specific language. This distinction allows clear separation of concern in development, a developer familiar with GF can implement the concrete part while someone familiar with the language targeted can fill out the lexicon without having to have much knowledge of GF. When adding a new language, the GF compilers makes sure that no rules in the abstract syntax goes unhandled in the concrete part and thus there is no risk of support for different languages diverging. The unified output despite input language differences helps reduce complexity of the overall system as the next module only has to handle abstract syntax trees as input.

3.3.4 Usage of GF in our prototype

GF has rudimentary support for Java in terms of a module named PGF (Portable Grammar Format) which has Java-bindings. However we could not use this for the prototype. The reason for this is that PGF has no executable for download and the safety precautions at the company did not allow the compilation steps to follow through which forced us to create a workaround. We have an interface in our Java-code as seen in figure 3.14, which we have then hidden part of our workaround behind.

In the concrete implementation of the interface for “getAbstractTreeAsString” we create a script file which we invoke the full GF runtime on, collect the result from and return to the main application. A similar process is done for the other methods with different script content. After invoking our GFTool implementation we get the abstract tree as a string which is used to build a parse tree representation in Java.

```
public interface GFTool {
    public String getAbstractTreeAsString(
        String input,
        String grammar) throws IOException;

    public String getTranslation(
        String tree,
        String fromGrammar,
        String toGrammar) throws IOException;
}
```

Figure 3.14: Interface for GF tool

If it would become possible to run PGF the only update is to replace the concrete implementation with one that uses PGF.

It is possible to export datatypes for many languages from GF including Haskell, JavaScript and Prolog among others but not Java at the moment. To gain as much help from GF as possible and ease development we have created our own tool which generates Java enums based on the abstract GF-grammar files including code to convert strings to enum instances. This allows us to convert the strings output from GF into datatypes native in Java and well supported by Java’s syntax, such as switch-statements while keeping coherence and reusing the grammar previously constructed. An example of such an enum instance can be seen in figure 3.15.

Creating this Java-code from the GF-grammar introduces a dependency on the parsing module in a later stage in the pipeline which is undesirable. While the overarching goal is to have the modules be as independent as possible, introducing such dependencies was considered to help development in a meaningful manner enough to justify its existence.

3.4 Interpreter

The interpreter is the part of the system that makes sure that the right command is executed and checks whether it is legal or not. Previous parts of the system lack context and is mechanically doing the job given the provided input. The interpreter, however, is aware of the current state of the system, and even though a command is legal with respect to the grammar, it might not be a valid command to execute given the current state of the system. This may be something simple as an ID not corresponding to any valid target, or ambiguities that can make one input acceptable for several commands.

If the system asks the operator a yes-and-no question, the answer is acceptable for any yes-and-no question, but should only execute commands responding to the current context. For example “Should I display information of target P22?” and “Should I stop tracking target P22?” can both be answered with “yes” but depending on which question was asked different actions should be taken by the system. Furthermore, a phrase like “yes” might be uttered at a time when no question was

```
public enum StatusParam {
    READY("Ready"),
    UNREADY("Unready"),
    UNARMED("Unarmed"),
    ARMED("Armed");

    private String functionString;

    private StatusParam(String s) {
        functionString = s;
    }

    public String getFunctionString() {
        return functionString;
    }

    public static StatusParam getParam(String s) {
        for (StatusParam p : StatusParam.values()) {
            if (p.getFunctionString().equals(s)) {
                return p;
            }
        }
        return null;
    }
}
```

Figure 3.15: Enumerator example from StatusParam

```
public interface Interpreter {  
    public void executeInstruction(Instruction instruction);  
}
```

Figure 3.16: Interface for Interpreter provided with instructions recognized

asked as well, the interpreter should then disregard the input since no question was asked and the statement is thus illegal.

If there is more than one possible parse tree but only one legal interpretation in the current context, only the command corresponding to that interpretation is to be output. If there is more than one valid interpretation the system handles it by prompting for further input to help disambiguate or ignore the command given. However, our proof of concept grammar does not include any ambiguities and Sphinx gives only one hypothesis when using grammars which saves us from this problem. Adding the interpreter steps helps improve precision since valid parse trees that are not legal are filtered away.

3.5 Integration

Alongside creating our own demo application we have been collaborating with the company to integrate it with its existing product. For us it was a simple process of exporting a runnable jar-file from our application and pointing out the correct class to instantiate. From the company side all that was necessary was to implement the interface seen in figure 3.16 with appropriate actions corresponding to instructions provided and instantiate our recognizer providing mentioned interface.

Such integration was successful and simply adding our prototype to existing projects for demonstration purposes was done with just a few hours of work, most of it consisting of mapping instructions to the existing product's actions. This integration was done with a predetermined command set agreed upon together. In order to introduce new commands a new runnable jar must be exported with proper enums generated by our toolchain.

3.6 Summary of pipeline

To revisit our initial presentation of the architecture but on a more concrete level see figure 3.17. Here we can see respective part of our pipeline and a representation of their input and output along with the example previously presented of selecting ready surveillance units.

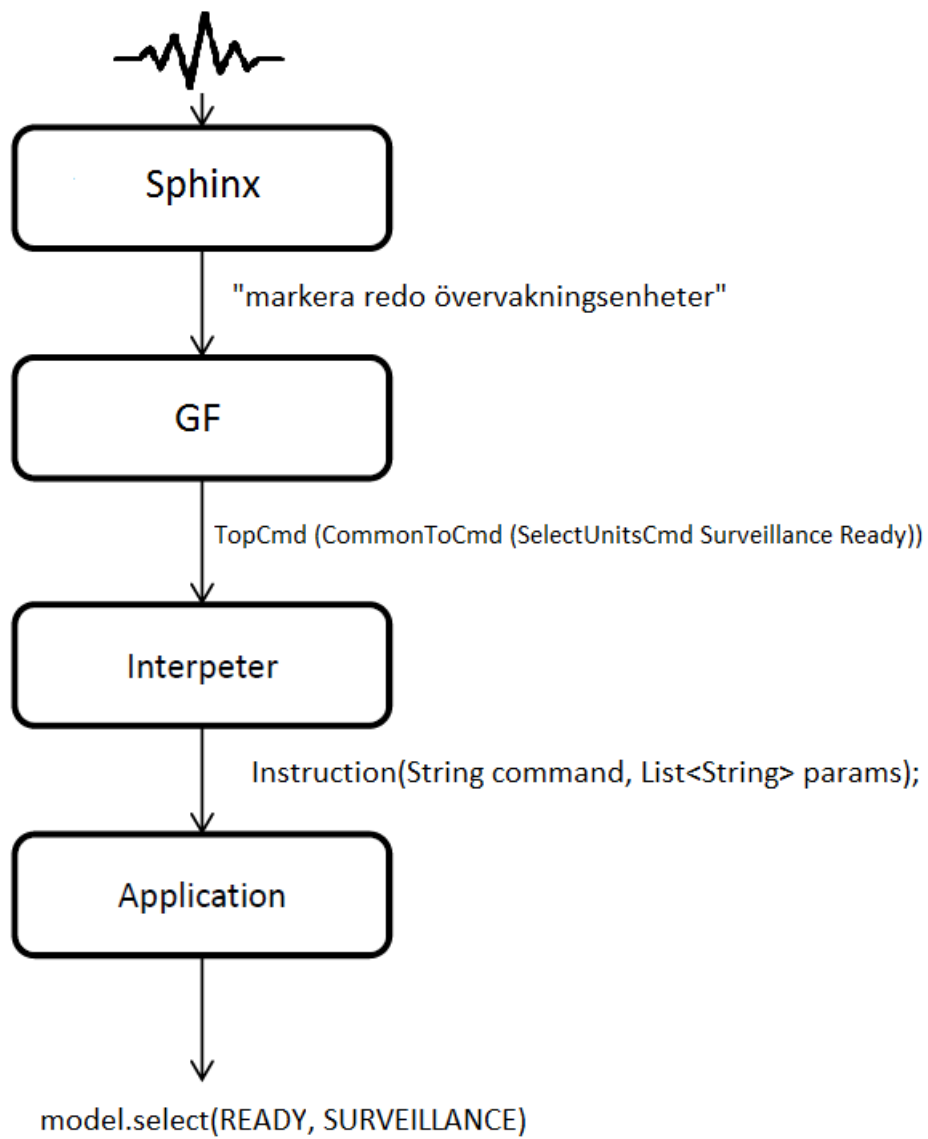


Figure 3.17: Concrete overview of prototype pipeline

3.7 Multilinguality

To have a system that can be modular enough to change language is a challenge. Not only do different languages use different words but the syntax differs as well. Sphinx already has acoustic models for the major languages, and it has support for creating new models for languages not yet supported by the community. This allows us to have a prototype as a proof of concept as well as keeping the possibility for adding new languages if desired in the future. Sphinx is used as the speech-to-text module in our prototype and we have used English and Spanish as proof of concept languages.

To have the input in text format does not solve the syntactical or semantical problem, GF is used to combat this problem. Due to the separation between the abstract and concrete grammar, the implementation of new languages is simple and flexible regardless of the syntax of the language. The semantics of what was said is extracted to an abstract form, which will look the same no matter which input language was used. This means that the system modules after the parser in the pipeline do not need to change if a new language is implemented, which helps minimize the effort needed to implement a new language.

3.8 Model adaptation

Sphinx and its toolchain have support for what is called acoustic model adaptation [29]. Instead of training a new model for a specific use case it allows adding training data and tweaking an existing model, adapting it to the situation at hand. We realized after some initial testing that the available US-English model provided by Sphinx does not perform to our expectations when it comes to native Swedish speakers and their English accents, and has thus explored the model adaptation feature. We gathered volunteers who are native Swedish speakers and recorded them speaking English phrases from our grammar used for demonstration purposes. We then used these recordings to run the model adaptation process on the existing US-English model provided by Sphinx.

4

Evaluation

The evaluation of the system has three main parts, the criteria on which the system was evaluated, the experimental set up when gathering results as well as the results.

4.1 Criteria

The evaluation of the project can be based on several criterias. We have chosen a subset to focus on which are the ones most in line with the aim of the project that can be measured easily. Each such criterion is listed and explained in this section.

Multilinguality

How well does the project supports multiple languages? We have chosen to measure this by how many changes we had to make in order to add a second and third language to the project and approximately how many hours of work are needed to incorporate the current set of available commands in a new target language. For example, if we support English, how difficult is it to add Spanish support? How many lines of code must be added and is there any other work required such as documentation, gathering training data etc? This comes from the aim of having a system that is as easy as possible to adapt to a new language in order to be viable from a business perspective.

Command/sentence error rate

How well does the system understand the user? How many sentences are correctly recognized by the system? We measure the systems precision by letting different users say two sets of sentences, one set of correct sentences and count correct interpretation and one set of incorrect sentences and count false positive interpreted sentences.

Gain from model adaptation

We have measured the command/sentence error rate with both respect to an adapted model and an original model. The adapted model comes from using Sphinx model adaption feature as described in section 4.2.

Usage complexity and instructions required

How difficult is it to become proficient in using the system? How much time is needed to become skilled in using the system? We measure this by the amount of instructions needed to understand how to operate the system as well as a qualified approximation on how long it takes for a new user to get used to using the system.

Runtime computer resource requirements

What resources does the project utilize while running? Due to the restricted available resources in the target environment for the project we have chosen to monitor and measure the resources usage in terms of CPU, memory and storage. All this with respect that responses should be delivered in real time by the system, in other word with as small delay as possible before an action is carried out.

4.2 Experimental setup

The experiment was done in English, since it is the most commonly known language other than Swedish at the company, and there is an acoustic model available for Sphinx in English.

4.2.1 Training step

First we performed a training step using Sphinx model adaptation due to the reasons explained in section 3.8. The volunteers consisted of 17 people, 13 men and 4 women. They were recorded saying the same 25 commands each resulting in audio data of just below 2 minutes per speaker and a total of roughly 30 minutes. This data was then used to adapt the model.

For the recording of the data and to simplify the work of creating the correct file structure a small program was created, whose GUI can be seen in figure 4.1. It presented the commands one by one to the user and allowed them to re-record commands if they felt it necessary. For each user it created a folder with appropriate file structure with sound files and transcription files. An example of the content of a transcription file can be seen in figure 4.2. This reduced the work for development as the file structure required by Sphinx was already present when it was time to run the training tool.

Once all training data was collected the training was done by running Sphinx adaption software and took just a few minutes. The adaption software takes an existing acoustic model as input and updates the parameters in accordance to the new training data. It updates the different probabilities in the underlying HMM to better map input sound to appropriate phones.

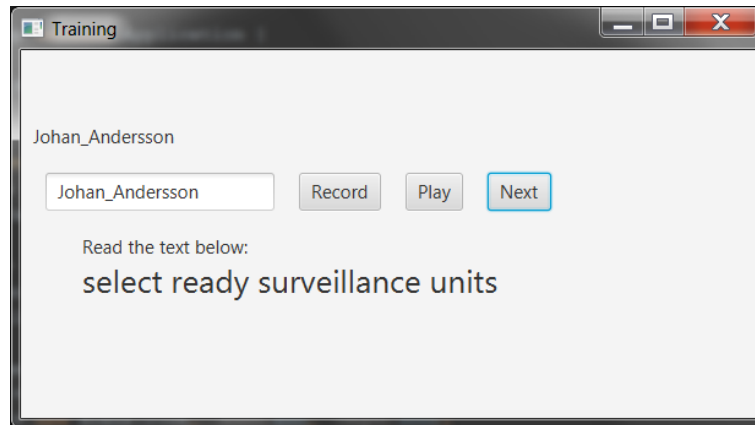


Figure 4.1: Screenshot of the simple training program that was developed

```
<s> select ready surveillance unit </s> (Johan_Andersson\001)
<s> give me the velocity of papa one one </s> (Johan_Andersson\002)
<s> select ready surveillance unit </s> (Oskar_Dahlberg\001)
...
```

Figure 4.2: How the transcription file could look like to adhere to Sphinx requirements

4.2.2 Evaluation step

Out of the original 17 volunteers, 10 had time to return for the evaluation step, 8 men and 2 women. The sentence error rate was then tested with both the adapted model and the original model to see if there is any difference in the result between the two runs. In order to minimize the bias of the result, 50% of the participants had the adapted model on the first run and the original on the second run and vice versa.

The test consisted of 24 legal phrases in order to check for an error rate, and 12 incorrect phrases of varying similarity to legal commands to check for false positives. Both the experiment and the recording of training data took place in a silent room with little interference from background noise. The process of the evaluation program was monitored.

4.2.3 Software used

In order to recreate the experimental setup, the tools and software are listed below:

- OS: Windows 7 Enterprise 64 bit
- Sphinx version: CMU Sphinx 4-5prealpha, released 2016-05-31
- Sphinx model: English, US, 5.2
- GF version: GF-3.8 for Windows
- Resource monitoring: Process explorer v16.2

4.2.4 Hardware

For hardware the following was used:

- Microphone: Konig electronic, serial number KN-MIC50C
- External sound card: Goobay, model number 68878
- Laptop: DELL laptop with i7-6820HQ CPU @ 2.70Ghz and 16 GB of RAM.

4.3 Results

Here we present the results gathered from our evaluation step described in section.

4.3.1 Implementing a new language

For the speech-to-text part we downloaded the community model for US-English and adapted it as described in section 3.8. It does work out of the box but we wanted to increase performance as well as evaluated Sphinx model adaptation features viability. For the second language our original plan was to have Swedish and believed we could reuse resources created at KTH previously [30] by converting those models to that of Sphinx format. There were tools available for such conversions but we did not manage to run them successfully on the existing Swedish models. Since the priority of the project was not to create acoustic models we moved to Spanish as our second language for the speech-to-text part.

For the parser module we started with an English version of our abstract grammar, the second language implemented was Swedish. This was done by creating a lexicon instance for Swedish and going back to the partial concrete grammar to correct a few mistakes. Those mistakes were not noticeable when using English but became apparent when the Swedish sentences were not those expected after adding a Swedish lexicon instance. All in all the time spent on introducing Swedish support for the parser took less than an hour. The reason we started with Swedish after English for the parser is partly due to it being our original target second language as well as our fluency of the language.

The third language added to the parser was Spanish, which took a bit more time since additional personnel that was fluent in Spanish had to be called in. In terms of development, time required when someone familiar with the language was present was comparable to the Swedish implementation; done within a very short timeframe of a few hours.

Overall, implementing a new language proved to be very easy if there is an acoustic model available. The original incomplete concrete grammar took some effort to create, mostly due to the need to familiarize ourselves with GF and the helper library. However, once the incomplete concrete grammar was present creating lexicon files were a trivial task done in short time for anyone familiar with the language to be implemented and did not really require any special knowledge of GF to be completed.

As can be seen in figure 4.1, the JSGF code generated by GF requires fewer lines of code for each individual language, but once the number of languages increases, the code required to make a new language work in GF is much less since

	GF	JSGF
Abstract/Concrete	239	-
English	92	143
Swedish	92	211
Spanish	92	279
Total	515	633

Table 4.1: Number of lines of code required for a new language in GF and in JSGF

the language specific rules do not have to be implemented. These numbers does not take into consideration the new lines GF generates in order to increase readability when converting to JSGF format, or the comments and new lines we have added in order to increase readability in the GF code.

In summary, to implement a new language in our pipeline there are three steps that must be carried out. Firstly, obtain an acoustic model for the new language. This can be done either by using an existing model created by the community or creating a new one. Secondly, create a lexicon file adhering to the abstract and incomplete concrete grammar present. In practice, this means creating a file of around 100 lines of code with domain specific words for the language in question. Thirdly, there is a compilation step and preparation of grammar for JSpeech Grammar Format. This is only a matter of running a tool in our toolchain and pointing it to the correct grammar; the tool compiles the grammar using GF, cleans the output and presents it in a Sphinx-compatible format. Performing these three steps can be done within half a workday, given that there is an acoustic model available.

4.3.2 Runtime resources

The acoustic models come compressed in an efficient binary format making them have a very small storage footprint. However, when running the application the memory usage is noticeable. Sphinx has less resource intensive models at the cost of performance loss which may be interesting in an even more restricted setting.

The CPU usage is the average usage during command intensive sessions with users providing commands almost constantly spread across many cores. Sphinx itself is single threaded; however we run the rest of our prototype and modules on separate threads from Sphinx in order to not block the recognition process needlessly. The parsing and such is thus run on separate threads.

4.3.3 Experimental results

The result from the experiment is quite interesting. As can be seen in table 4.3, there was an improvement in the hit rate by about 8%. What can also be noted is that there is a large difference between the two groups performance and their first and second round. The group that started with the non-adapted model had an error rate of 41% using the non-adapted model and 27% on the adapted model, which is

	Resources used
CPU	7,79%
Memory (RAM)	1.08 GB
Storage (US-EN model)	52.6 MB
Storage (Adapted US-EN model)	52.6 MB
Storage (Spanish model)	46.9 MB
Storage (GF source + compiled)	< 1 MB

Table 4.2: Resources required by prototype

	Non Adapted Model	Adapted Model
Adapted first	0,25	0,24
Not adapted first	0,41	0,27
Total	0,33	0,25

Table 4.3: Sentence error rate for the adapted model and the original model. For all participants combined and depending on which model they did first.

a huge improvement. The group that started with the adapted model had an error rate of 25% using the non-adapted model and 24% on the adapted model.

What else could be noted from this was that the persons whose results were good in the first round, had good result in the second round as well, while the individuals whose results were poor in the first round generally increased the performance regardless of which model they started with.

The result of false positives are around 27% for both models and can be seen in table 4.4. False positives are sentences that are not legal in the system but were parsed into a legal command regardless. The majority of the false positives come from phrases which share much of their structure with that of a legal command. For instance, the phrase “give me the velocity of yada yada” would be legal if the words “yada yada” would be replaced with a valid ID, and it was parsed into a legal ID almost every time. A sentence that was completely different from a command, like “order me some McDonalds” or “how are you”, did rarely parse into a legal command, though that did happen on occasion. Phrases like those are more likely to be uttered to another person when talking rather than something that resembles a command.

An important aspect that must be pointed out is that the amount of result gathered is a bit low to draw confident conclusions about the performance. There are only 10 people present for the evaluation step with limited time which means that performance may be better or worse than indicated in these results.

4.3.4 Usability

In terms of our prototype it was used in two cases, firstly a small integration with existing system mapping voice commands to existing GUI-shortcuts, secondly our own demo application of a radar system with some more elaborate voice commands to select unit types etc. The shortcuts that were added to the real application might

	Non Adapted Model	Adapted Model
Adapted first	0,32	0,27
Not adapted first	0,25	0,27
Total	0,28	0,27

Table 4.4: False positives for the adapted model and the original model. For all participants combined and depending on which model they did first.

not add too much to the experience. Features like zooming on the map shown or hide and show panels are easier to do with the shortcuts at hand. The reason why those commands were chosen was due to that fact that they were shortcuts in the existing system and were easy to implement. The real improvement was more easily seen when implementing the demo environment and the operator is able to select interesting targets or show certain information without having to open panels in order to execute the command, in other words more complicated actions.

The commands chosen might not have been the most intuitive ones in that they shared similar traits which could lead to confusion of which command used what words.

5

Discussion

5.1 Discussion

In this section we present our thoughts on the project and the outcome of it.

5.1.1 Grammatical Framework

GF has been a good tool to use when handling multiple languages. Due to the abstract grammar and concrete grammar in combination with the support library, it has been easy to implement new languages. Even though the first language was a little more work to implement in GF than another grammar format dedicated for that specific language with the same purpose, the second language was implemented in less than an hour which is remarkable little effort for localization to a new language. It does take some effort to learn to navigate the structure and become efficient as with any new programming language, especially when it comes to how to construct grammatical rules using the helper library in an efficient manner. In this case we consider the effort well worth the gain that GF provides us.

5.1.2 Experiment discussion

There is room for improvement when it comes to how the experiment was carried out. However, the results indicate an improvement in accuracy and some of the participants commented on that they experienced that the adapted model did work better. Sphinx model adaptation feature does work, even on a relatively small amount of data. Comparing our training data of just 60 MB compared to training a new acoustic model which may require over many GBs of data depending on use case.

It is worth noting that the sample size of the training and evaluation were rather small, this is due to limitations regarding bringing equipment and data out of the company and bringing external people into the building for security reasons. It was deemed to be easier to use employees already working on site to help with the evaluation. Thus only employees who had time to spare could help with the experiment and training of the model. However, we feel confident that the results point to an improvement using Sphinx model adaptation as well as the usability of our proposed architecture overall.

Regarding the result of the model adaptation, there was a tendency that the second round generally scored higher, there could be several reasons why this was the case. The first round contained more hesitation in the speech from the volunteer,

which became better as more commands were said. In the second round the phrases had already been said once by the volunteer, thus less effort had to be put into reading the same sentence again and understanding the structure. Possibly, this resulted in better pronunciations and less hesitation.

As for the result of the experiment, even though there was an improvement in the accuracy after adapting the existing model, an error rate of 25% when listening is not good enough to be used reliably in the end system. The top performer had an error rate around 5-10%, however, the data set was not large enough to draw any definite conclusions. There are a few methods that could combat the low error rate, first is to give the system more training data in order to recognize accent of the user. The question is how much training data has to be gathered in order to have the accuracy to be at an acceptable level.

Another solution that Sphinx offers is to interpret the content of a file, since there is a finite amount of data, the recognition can be on the entire file rather than guessing when the person is done talking. This could increase the speech-to-text accuracy. If neither of these methods would prove to be accurate enough, there might be a good idea to start looking for another product that translates speech-to-text. The reason why we chose Sphinx is partly due to its free license, the fact that it has support for grammar and the low effort required to set it up in order to run the system. With more time put into the project, a better speech-to-text engine might be implemented and could improve the performance.

5.1.3 Precision

The area of the project we are least content about is the precision of the prototype. While it does perform well for some users it fails often enough so that it becomes an annoyance. We do believe this could partly be remedied by collecting more training data for adaptation with a better suited test group and perhaps revising the recording method. As of the moment the recording is done continuously and Sphinx tries to determine when a user starts and stops speaking a command. To our knowledge Sphinx does not expose any method to tweak the thresholds for such detections and depending on environment this detection works more or less well. Throughout the project we noticed it is very common for Sphinx to cut off input and starting analysis prematurely, especially if there is any hesitation in the speech from the user. It would be interesting to compare our result with using a push-to-talk method to record a whole command and then apply Sphinx recognizer with our model to see if the result would be better.

5.2 Critical system safety

A very important aspect is the reliability of the system. If the system and technology are to be used in close co-operation with armed forces it is important that it performs according to expectation and does not provide incorrect results. For example, a misinterpreted command from a user cannot be allowed to be the reason to put human lives in danger. It is important to be aware of possible problem and consider

it when developing such a system as ours and provide methods to deal with situations where potential incorrect results could have dire consequences.

In our case we did this by modeling our commands in two categories, common commands and sensitive commands. Common commands should require a lower threshold of confidence in order to be executed and may not require verification from the user. In this category we consider things such as GUI-updates. Sensitive commands we considered to be of such nature that a higher level of confidence is required for them to be carried out. They could require the user to verify that the interpretation is correct before continuing. An example of such a command would be turning the radar signal on and thus indirectly exposing the position to possible hostiles.

Another important aspect is accountability. What measures are in place to ensure that the system performs as expected? And if it does not, what data is collected and what traceability for the reason is present? It should always be possible to find out why a certain course of action was present in the system which was one of the reasons why we choose a modular approach with transparent steps with many possible points of checkpoints for the result produced.

There are likely many more considerations to take into account, especially when trying to introduce a system like ours into a real world application. However the ones previously mentioned are those we consider important to highlight and be aware of at least in early stages of development of an ASR system in a domain such as ours.

5.3 Future work

Something to consider for the future is the choice of words used in the commands. During the implementation we noticed that Sphinx had problems distinguishing words that have similar characteristics. One command had three different states but with only one word differencing. The command could be *select*, *accept* or *remove*. Sphinx had some problem separating the words *accept* and *select*, since they have similar structure. We solved this by changing the word *accept* to *approve*, which has the same meaning but different word structure. Sphinx could then easily distinguish *select* from *approve*. However, now there was a problem with *approve* and *remove* since these words share similar traits. We eventually solved this by changing *remove* to *decline*, but this is a problem that likely will emerge again. If continued work is done on this prototype and more commands are implemented, the choice of words will have to be more carefully considered in order to avoid similar words that might increase the risk for errors when interpreting speech.

To increase the precision of the system there are a number of different solutions that could be tested. For our project we have tested one speech-to-text system. It is possible that there are other alternatives that perform better with the same functionality. We did not look much into commercial alternatives or Windows only systems, which may be an option for our company partner.

We do recommend that if further explorations of other alternatives to Sphinx are looked into that support for grammars are considered a high priority. Since grammars have been noted to increase precision [4] and offer better control for the developer which is useful in a restricted domain.

If continued work on Sphinx is performed our recommendation is to see if an alternative way of handling input data would help. One of the common errors we encountered is that Sphinx cuts off input and starts performing analysis before the user has finished speaking. This could possibly be remedied by changing the input method from continuous to a push-to-talk approach or similar where the whole command is first recorded and then supplied to Sphinx, guaranteeing that the full input is used as basis for the interpretation. This should not be very difficult or time consuming to implement.

Furthermore, it would be interesting to see if a better test group with people comfortable in using an ASR-system and the domain would record training data and in a larger extent than 30 minutes. A good starting point would probably be 20 people or more recording 5 minutes of commands present in the domain to be used, resulting in a total of 100 minutes of recorded data. The evaluation of the result should also be given more room with a larger amount of commands evaluated. 50 commands instead of the 24 used here to better assess the outcome of the training.

Bibliography

- [1] *Speech API - Speech Recognition | Google Cloud Platform*. <https://cloud.google.com/speech/>. Accessed: 2017-05-10.
- [2] *Alexa Voice Service*. <https://developer.amazon.com/alexa-voice-service>. Accessed: 2017-05-10.
- [3] Björn Bringert. *Programming language techniques for natural language applications*. Department of Computer Science and Engineering, 2008.
- [4] Beth Ann Hockey and Manny Rayner. “Comparison of grammar-based and statistical language models trained on the same data”. In: *Proceedings of the AAAI Workshop on Spoken Language Understanding*. 2005.
- [5] Automatische Spraakherkenning, Wiggers, and LJM Rothkrantz. “Automatic Speech Recognition USING Hidden Markov Models”. In: (2003).
- [6] Stanley S Stevens and John Volkman. “The relation of pitch to frequency: A revised scale”. In: *The American Journal of Psychology* 53.3 (1940), pp. 329–353.
- [7] Daniel Jurafsky and H James. “Speech and language processing an introduction to natural language processing, computational linguistics, and speech”. In: (2000).
- [8] Naomi Sager. *Natural language information processing*. Addison-Wesley Publishing Company, Advanced Book Program, 1981.
- [9] Lawrence R Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286.
- [10] Stephen Tu. *Derivation of baum-welch algorithm for hidden markov models*. 2015.
- [11] *CMU Sphinx US-En Dictionary*. <https://github.com/cmuspinx/sphinx4/blob/master/sphinx4-data/src/main/resources/edu/cmu/sphinx/models/en-us/cmudict-en-us.dict>. Accessed: 2017-06-11.
- [12] R Thangarajan, AM Natarajan, and M Selvam. “Word and triphone based approaches in continuous speech recognition for Tamil language”. In: *WSEAS transactions on signal processing* 4.3 (2008), pp. 76–86.
- [13] Petra Perner, Patrick Wang, and Azriel Rosenfeld. *Advances in Structural and Syntactical Pattern Recognition: 6th International Workshop, SSPR'96, Leipzig, Germany, August, 20-23, 1996, Proceedings*. Vol. 1121. Springer Science & Business Media, 1996.
- [14] Danielle S McNamara. “Computational methods to extract meaning from text and advance theories of human cognition”. In: *Topics in Cognitive Science* 3.1 (2011), pp. 3–17.

- [15] Aarne Ranta. *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012.
- [16] *syntax - definition of syntax in english*. <https://en.oxforddictionaries.com/definition/syntax>. Accessed: 2017-04-06.
- [17] Jane Garry and Carl Rubino. “Facts about the World’s Languages”. In: *HW Wilson* (2001).
- [18] *semantics - definition of semantics in english*. <https://en.oxforddictionaries.com/definition/semantics>. Accessed: 2017-04-06.
- [19] *Double entendre examples*. <http://examples.yourdictionary.com/double-entendre-examples.html>. Accessed: 2017-04-10.
- [20] *JSGF Grammars in Sphinx4 [CMUSphinx Wiki]*. <http://cmusphinx.sourceforge.net/wiki/sphinx4:jsgfsupport>. Accessed: 2017-05-10.
- [21] Lothar Budach. *Fundamentals of Computation Theory: 8th International Conference, FCT’91, Gosen, Germany, September 9-13, 1991. Proceedings*. Vol. 529. Springer Science & Business Media, 1991.
- [22] Ian H Witten et al. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.
- [23] *HTK Speech Recognition Toolkit*. <http://htk.eng.cam.ac.uk/>. Accessed: 2017-05-16.
- [24] *Kaldi ASR*. <http://kaldi-asr.org/>. Accessed: 2017-05-16.
- [25] *CMUSphinx Open Source Speech Recognition*. <https://cmusphinx.github.io/>. Accessed: 2017-05-16.
- [26] *Kaldi grammar support*. <https://sourceforge.net/p/kaldi/mailman/message/31895432/>. Accessed: 2017-05-16.
- [27] *About - CMUSphinx Open Source Speech Recognition*. <https://cmusphinx.github.io/wiki/about/>. Accessed: 2017-05-16.
- [28] Aarne Ranta. *Grammatical framework: Programming with multilingual grammars*. CSLI Publications, Center for the Study of Language and Information, 2011.
- [29] *Adaptation of the acoustic model*. <https://cmusphinx.github.io/wiki/tutorialadapt/>. Accessed: 2017-05-15.
- [30] Giampiero Salvi. “Developing acoustic models for automatic speech recognition in Swedish”. In: *European Student Journal of Language and Speech, Stockholm, Sweden* (1999).