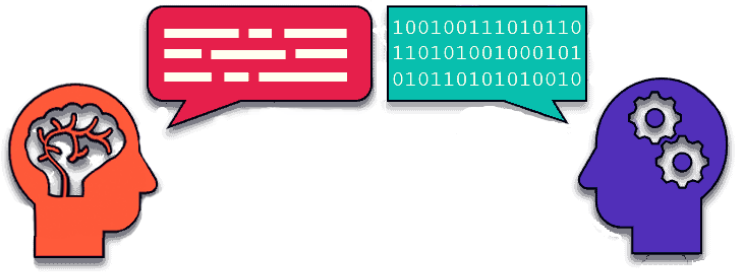
Bachelor Thesis

Computer Science and Engineering, 300 credits



Generation of Control Logic from Ordinary Speech

Halmstad 2022-03-01
Elias Vahlberg, Hamed Haghjo

HÖGSKOLAN
I HALMSTAD

Abstract

Developments in automatic code generation are evolving remarkably fast, with companies and researchers competing to reach human-level accuracy and capability. Advancements in this field primarily focus on using machine learning models for end-to-end code generation. This project introduces the system CodeFromVoice, which explores an alternative method for code generation. This method relies on existing Natural Language Processing models combined with traditional parsing methods. CodeFromVoice shows that this approach can generate code from text or transcribed speech using Automatic Speech Recognition. The generated code is limited in complexity and restricted to the context of an existing application but achieves a Word Error Rate of less than 25%.

Sammanfattning

Utvecklingen av automatisk kodgenerering visar stora framsteg, med företag och forskare som tävlar om att nå mänsklig nivå av noggrannhet och förmåga. Framsteg inom detta område fokuserar främst på användning av maskininlärningsmodeller för hela kodgenerering processen. Detta projekt introducerar systemet CodeFromVoice, som utforskar en alternativ metod för kodgenerering. Denna metod bygger på befintliga NLP-modeller kombinerat med traditionella parsning metoder. CodeFromVoice visar att detta tillvägagångssätt kan generera kod från text eller transkriberat tal med automatisk taligenkänning. Den genererade koden är begränsad i komplexitet och begränsad till sammanhanget av en existerande applikation, men uppnår en ordfelfrekvens som är mindre än 25 %.

Acknowledgment

This project is in collaboration with HMS Networks AB, and we would like to thank the HMS Labs team for guidance and support. From this team, we would especially like to thank Felix Nilsson for helpful input, discussions, and guidance throughout the project. We thank our supervisor Hadi Fanaee (Halmstad University), for providing good feedback.

Contents

1 Introduction

Generating code from natural language is a complex task due to the vast amount of ambiguities in natural language compared to syntactically unambiguous code. Many approaches have recently emerged to tackle this task because of advancements in Machine Learning (ML) and Natural Language Processing (NLP) [1]. Much of the research is focused on generating more complex code structures with less context-specific interpretations. The training of such models is possible mainly because of "Big Code," which refers to the increasing amount of available source code from open source repositories and forums such as Stack Overflow.

This project proposes an alternative approach to the problem of code generation that forgoes the need for large datasets and extensive training. It does this by assigning syntactic structure and subdividing sentences into constituent parts as a dimensionality reduction method to allow for deterministic parsing. Named Entity Recognition (NER) extracts information not dependent on the syntactic structure, such as class and function names. The parsing methods in Section 4.3.2 and 4.3.3 generate the code.

1.1 Delimitations

The scope of this project includes:

- Development of a Logic Parsing System and NER System detailed in sections 4.3.3, 4.3.2.
- Integration and configurations of the NLP system CoreNLP and automatic speech recognition system Vosk [2, 3].
- The creation of evaluation data for evaluating each processing step.

System delimitations:

- One input sequence can't contain multiple sentences.
- The output code is limited to the set of terminal symbols shown in Appendix A.2

- Functions, classes, and fields must be annotated and available to the system for them to be used shown in Appendix A.3.

Report delimitations, the report will not contain :

- A usage guide for CodeFromVoice.
- A comprehensive overview of topics relating Computational linguistics and Artificial intelligence. *The sections describing these topics aim to make later sections more comprehensible. Relating concepts and methods used and their relation to this project.*

1.2 Task specification

The task specification remains primarily unchanged since the start of the project. However, the method of parsing input text has changed from an attempt to utilize the cross-lingual syntactic representation generated by UDepLambda [4] to parsing the constituents of the given sentence, which gives more freedom to utilize constituents, Part-Of-Speech (POS) tags, and named entity references to produce a logical meaning representation (LMR) that could be utilized for code generation. Due to several ambiguities in speech, such as different dialects and the lack of punctuations for written text, the evaluation metrics Exact Match (EM) and Word Error Rate (WER) is used to measure the system's accuracy. The goal for WER is <25% due to how dependent EM is on the type test, it is used as a secondary metric, but no goal is set for this. The motivation for these goals is that the system would be considered viable in the various use cases, with minimum need for human interventions in any of its parts.

1. Take speech as input and convert it to text
2. Punctuate the text
3. Identify and map words to existing named entities
4. Generate a LMR of the given text
5. Translate the generated LMR using a set of rules into executable code linked to an existing program
6. Demonstrate that tasks 1-5 produce correct results that fall in the range of the set goals

2 Background

This chapter introduces Natural Language Processing, Automatic Speech Recognition (ASR), and Code Generation. A short discussion of related work then follows this.

2.1 Natural Language Processing

Natural Language Processing is an interdisciplinary field in computer science, artificial intelligence, and linguistics covering the activities involved in modeling human language, managing natural languages, recognizing natural language, and producing natural language. Therefore, NLP is related to the area of human-computer interaction.

Researchers are putting much effort into NLP because the demand is growing. Especially with the growing popularity of Voice User Interfaces (VUI) [5]. Some notable examples of the most popular VUI's include Siri (Apple's virtual assistant), Google Assistant, and Amazon Alexa [6].

NLP's two main components are Natural Language Understanding (NLU) and Natural Language Generation (NLG). This thesis primarily focuses on NLU.

2.1.1 Natural Language Understanding

Natural Language Understanding is the task of enabling computers to derive meaning from human or natural language input. It is a subfield of artificial intelligence, computer science, and linguistics. For example, Question Answering (QA) algorithms fall under this category.

The main challenge in Natural Language Understanding is the ambiguity of human and natural language, the need for precise information, and the different languages and dialects people speak [7]. Natural language is ambiguous by nature, so there is often more than one way to interpret what a person says. The same sentence can have different meanings depending on the context, tone, and body language. For example, "He's not here" can be interpreted as a statement of fact, an apology, or a question.

Many interpretations and ambiguities must get resolved to generate a deterministic code.

2.2 Automatic Speech Recognition

Automatic Speech Recognition (ASR) uses machine learning methods to recognize speech using a digital audio input instead of a keyboard or mouse. Computer software trained to recognize individual speakers or classes of speakers based on their speech patterns performs the automatic speech recognition process. In a supervised learning paradigm, humans typically provide transcripts of labels with audio data for training. The goal is to develop algorithms that can predict labels given an audio input.

2.3 Code Generation

Source code modeling and generation is the process of predicting explicit code from data containing indications of what the code does or how it behaves [1]. One can make predictions using data such as incomplete code, code in another programming language, or natural language descriptions [1]. This project focuses on code generation from Natural Language descriptions.

Because Natural language exhibits ambiguities in both structure and meaning, it is not comprehensible by a computer [8]. Different approaches exist that reduce ambiguities and restrict the scope [1]. Most data-driven methods treat this problem as a language generation task relying on the neural network models to remove ambiguities [9]. The system proposed in this thesis uses the syntactic structure of natural language to reduce ambiguities and the NER system to restrict the scope.

2.4 Related Work

This section details some of the current state knowledge in code generation and semantic parsing. These differ from the approach described in this thesis but serve as a good overview.

2.4.1 TranX

TranX is a non-commercial transition-based neural semantic parser. TranX maps natural language utterances into Abstract Syntax Trees (AST) or python source code [10]. The difference between TranX and the system proposed in this thesis, CodeFromVoice, is that CodeFromVoice is context-dependant and uses the constituency parsing from CoreNLP, and TranX is trained as a model instead of running on a set

of rules to process the input [10].

2.4.2 Amazon Alexa

Amazon Alexa is a commercial cloud-based voice service that provides several capabilities such as playing music, answering questions, and providing weather and traffic reports [6]. Like the system proposed in this thesis, Alexa uses automatic speech recognition (ASR) to convert utterances to text and Natural Language Understanding (NLU) to determine the intent [6]. Alexa differs from the system described in this thesis because it does not generate any control logic and relies on pre-existing control logic already embedded in the devices [6]. It also uses machine learning to personalize itself to its current environment.

2.4.3 OpenAI Codex and GPT-3

GPT-3 is an autoregressive language model, the third generation of Generative Pretrained Transformer (GPT-3) [11]. GPT-3 contains 175B parameters, 96 layers, and was trained on a corpus of 499B tokens of web content [11]. This makes GPT-3 the most extensive language model constructed [11].

Codex is a descendant of this transformer model. Its training data contains, besides natural language, even billions of lines of source code from publicly available sources, including code in public GitHub repositories [12].

Codex takes a natural language description of a programming problem as input. Codex can generate codes in most common programming languages as output from this. Codex vastly outperforms all OpenAI's previous models and most competing ones too [13]. However, it is more computationally demanding and requires an unprecedented amount of training data [13]. The system described in this report does lack a great deal of functionality and complexity compared to Codex but requires much less processing time.

3 Theory

3.1 Neural Networks

Neural Networks are a family of machine learning algorithms inspired by the structure of the human brain. The goal of artificial neural networks is to simulate the human brain's cognitive functions like learning and memory by modeling neural structures with mathematical functions [14]. Neural networks can classify images or find patterns in large datasets. The most elementary form of a neural network is the Single-layer perceptron.

A Single-layer perceptron is a neural network used for binary classification. The perceptron takes in input, multiplies it by weights, and outputs a prediction. If the prediction is correct, the perceptron continues to the following inputs. If the prediction is incorrect, the perceptron adjusts the weights and then makes a new prediction [14].

3.1.1 Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of artificial neural networks. RNNs are defined by having recurrent connections, either for sets of neurons or layers [15]. A recurrent connection retains some information between feed-forward operations in their internal state see Figure 3.1.1. For this reason, RNNs are particularly suited for tasks that involve sequences, for example, in ASR. More specific RNN variants such as the Elman networks are used for structured prediction learning [16].

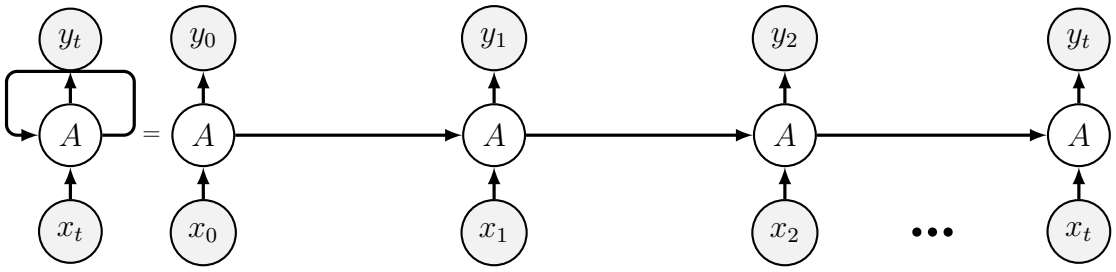


Figure 3.1: RNN with input node x, output node y and one recurrent connection.

3.1.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a "more sophisticated" version of the RNN. The difference between these and RNNs is the addition of the forget gate, which regulates the flow of information through a unit see Figure 3.2 [16]. During training, the LSTM introduced itself to solving the "vanishing gradients" and "exploding gradients" problems. Newer Sequence to Sequence (Seq2Seq) approaches now favor LSTM over older RNN approaches [16].

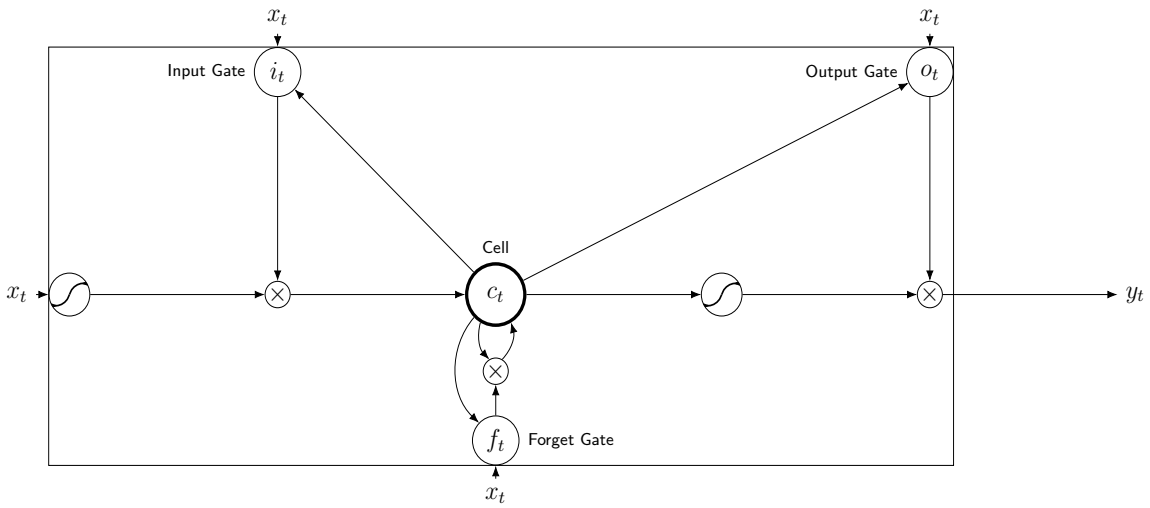


Figure 3.2: LSTM node diagram, with input x and output y [16]

3.1.3 Language Models

Language Models (LM) are neural network models (often RNN or LSTM) with the specific purpose of handling natural language or speech [17]. One widely used model design is the Maximum Entropy (ME) RNN model (RNNME) [17]. The RNNME model bases predictions on the weighted sum of the underlying ME and RNN models. N-gram models are another type of language model which sees use in many NLP frameworks and are the most probable next-word predictions [18]. N-gram models are beneficial as a complement to an RNNME or LSTM model [18].

3.2 Automata Theory and Parsing

3.2.1 Automata

An automaton is an abstract model of a digital computer. Automata have a mechanism for reading input with a specific alphabet and a mechanism for writing output in the defined output alphabet [19]. The automaton must have a set of states. These states will be visited in a specific order depending on the input [19].

$$M = \langle \Sigma, \Gamma, Q, \delta, \lambda \rangle \tag{3.1}$$

$\boldsymbol{\Sigma}$: Input alphabet

$\boldsymbol{\Gamma}$: Output alphabet

$\mathbf{Q}$: Set of states

$\boldsymbol{\delta}$: Transition function $\delta : Q \times \Sigma \to Q$

$\boldsymbol{\lambda}$: Next output function $\lambda : Q \times \Sigma \to \Gamma$

Figure 3.3: Automaton definition [19].

The most basic way to represent an automaton is as a deterministic finite accepter, as shown below.

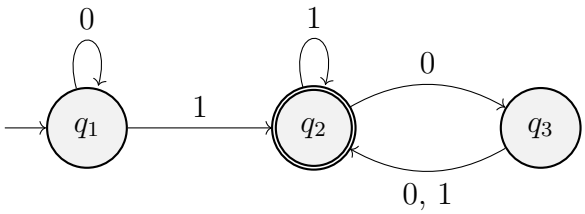


Figure 3.4: Illustration of a deterministic finite accepter [19].

3.2.2 Grammar

Grammar within the context of automata theory is the description of the rules for structuring a language. Grammar is also described as the fundamental language-

definition mechanism and is the fundamental basis for parsing. The common of these grammars is Context-Free Grammar (CFG) and is formally defined as:

$$G = \langle V, T, S, P \rangle \tag{3.2}$$

V: Finite set of variables.

T: Finite set of terminal symbols.

S: Start symbol ($S \in V$).

P: Finite set of production rules.

Figure 3.5: Context-free grammar definition [19].

Even though humans have been using informal grammar systems in natural language for thousands of years, there still exists no formalized grammar system that is capable of describing all parts of natural language deterministically [19]. Moreover, this would mean that a large number of linguistic structures still cannot be accurately represented without leaving room for error [8].

3.2.3 Languages

Defining what a language is can be pretty tricky. In this thesis, the symmetric group of all terminal symbols for a specific grammar [19] is the only language-definition mechanism used. The definition of Context-Free Languages (CFL) is as follows:

Let $G = \langle V, T, S, P \rangle$ be a grammar.

Let $w_1 \overset{*}{\Rightarrow} w_n$

denote the derivation of w_1 with an unspecified number of steps.
Then the set

$$L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$$

is the language generated by G.

Figure 3.6: Context-free language definition [19].

3.2.4 Parsing

Parsing is the process of finding a sequence of productions such that $w \in L(G)$ is derived [19]. The grammar that defines the target language determines the behavior of a parser. Many methods can accomplish this task. The simplest form of parsing would be to apply all leftmost derivations [19]. However, a language defined by such grammar would be sorely lacking in expressive capability though many languages follow simple grammar. The most recognizable of these would be the infix notation used in arithmetical formulas. When attempting to parse natural language unique methods of parsing such as probabilistic methods described in Section 3.4.1.

3.3 NLP Annotators

NLP annotators define what information should be extracted from any input sentence. The set of NLP annotators outline the language processing capabilities that is desired for a specific task. The NLP annotators that CodeFromVoice uses and is later mentioned are the following. Tokenization (tokenize) is splitting the text into individual tokens. Sentence Splitting (ssplit) is splitting the text into sentences. Part-of-speech tagging is the process of assigning part-of-speech tags to each token. Appendix Section A.3 gives a description to each POS-tag. Morphological Analysis (Lemmatization) is the process of reducing a word to its base form. For example, "walking" would be reduced to "walk". Named Entity Recognition is the process of assigning named entity tags to each token. Regular Expression Named Entity Recognition (Regexner) is the process of matching tokens against regular expressions.

3.4 Constituency Parsing

Constituency parsing is a task in natural language processing that involves identifying the constituents of a sentence, such as noun phrases and verb phrases. A constituency parser produces a graphical representation of the constituents of a sentence and their relationships to one another. There are many different algorithms for constituency parsing. This section will describe the parser type used by CoreNLP, which is a Probabilistic Context-Free Grammar (PCFG) parser [3].

3.4.1 Probabilistic Context-Free Grammar

A PCFG parser is a probabilistic context-free parser that uses a neural network to learn how to parse sentences [20]. A corpus of manually annotated sentences with their

constituent structures trains the parser. The neural network predicts the constituents of a sentence, given the words of the sentence, using the training data [20]. The definition of PCFG is as follows:

$$G_p = \langle V, T, S, P, Pr \rangle \tag{3.3}$$

Figure 3.7: Probabilistic context-free grammar definition [8]

The difference between PCFG and CFG is that PCFG is a quintuple and includes the Pr term containing the set of probabilities on production rules. This term implies that the parsing process becomes non-deterministic. Methods of increasing the accuracy include Cocke–Younger–Kasami (CYK) algorithm to find the optimal parse tree [8], grammar sequence alignment optimization [8].

3.5 Constituency Grammar

A syntactic constituency is a group of words that can behave as single units or constituents. Each type of constituent is identifiable by its tag and the sequence of words or the other constituents that make it up. The syntactic structure is defined from bottom to top, starting with words and phrases to form simple sentences, then building up to more complex grammatical structures such as clauses and compound sentences. A constituency parser extracts the constituency grammar into a syntax tree. It identifies the Parts-Of-Speech tags (POS tags), phrases, and clauses of the sentence until it can form a complete parse tree. The type of constituency parser CodeFromVoice uses is a part of CoreNLP. This parser uses the Penn TreeBank (PTB) tagset to label constituents. Appendix A.4 shows the complete list of tags and their corresponding meanings.

3.6 Semantic Parsing

One branch of NLP that is often useful in computer-human interaction is the possibility to parse natural languages into their logical meaning. This task is called semantic parsing and can be a complex problem due to the vast amount of ambiguities in natural language [21].

There are two types of semantic parsing, shallow and deep semantic parsing. Shallow

semantic parsing is concerned with grouping and identifying entities based on their semantic role. Deep semantic parsers focus on identifying parts of the sentence and disambiguating their roles in a more specialized way [21]. Unlike Constituent parsing, a form of syntactic parsing of sentences, semantic parsing is also concerned with identifying and grouping the parts of a sentence based on their logical meaning [21].

Both shallow and deep semantic parsing are related to the same task of extracting logical meaning from a sentence. However, shallow semantic parsing focuses on disambiguating terms to understand further how they relate to the sentence. On the other hand, deep semantic parsers are concerned with understanding the sentences containing significant compositionality [21]. The type of semantic parser developed in this project falls somewhere between these two types containing aspects of deep semantic parsing but following approaches used for shallow semantic parsing.

4 Method

4.1 Software used

4.1.1 ASR

To evaluate the effectiveness of the different ASR solutions, the datasets Librispeech test-clean corpus [22] and TED-LIUM corpus [23] are used. Librispeech test-clean corpus is approximately 1000 hours of English speech derived from audiobooks from the LibriVox project [22]. TED-LIUM corpus comprises 118 hours of English speech derived from TED Talks [23].

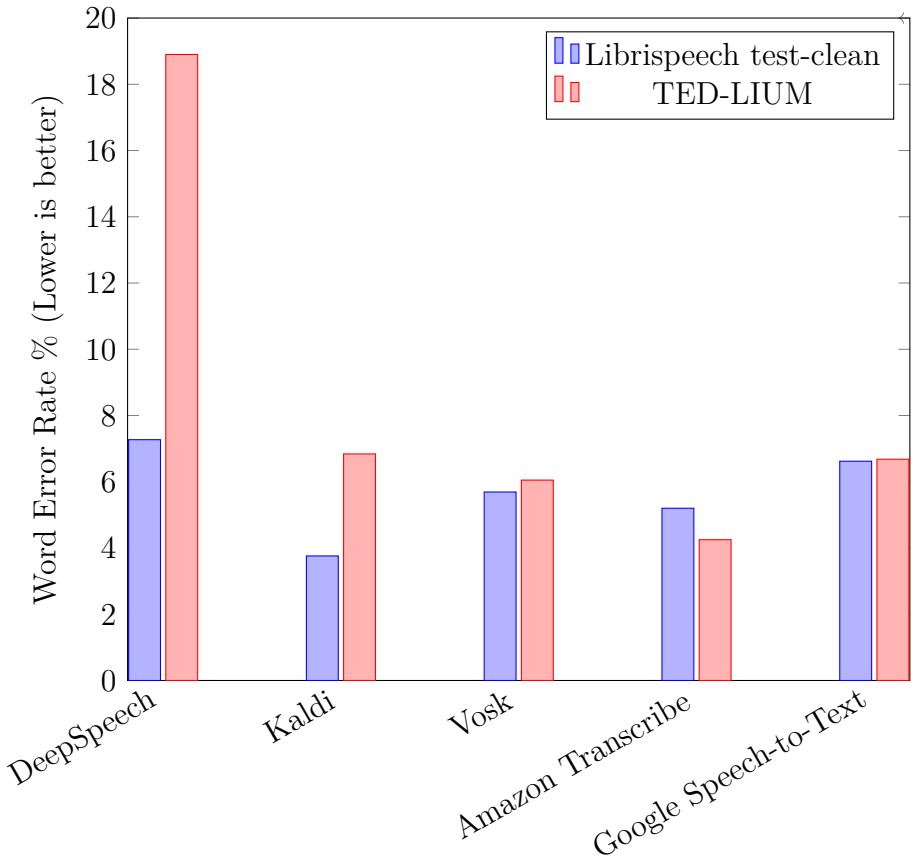


Figure 4.1: Speech-to-Text Benchmark [24–26].

Table 4.1: Speech-to-Text benchmark results (Accuracy), WER (Lower is better)

Engine	**Librispeech test-clean (% WER)**	**TED-LIUM (%WER)**
Mozilla DeepSpeech [24]	7.27	18.9
Kaldi (model m13,m5) [25]	3.76	6.84
Vosk (Accurate generic English Model) [26]	5.69	6.05
Amazon Transcribe [24]	5.20	4.25
Google Speech-to-Text (Enhanced) [24]	6.62	6.68

As shown in Figure 4.1 and Table 4.1 the performance varies significantly with Mozilla DeepSpeech displaying much higher WER in both the Librispeech and TED-LIUM benchmark 4.1. However, both Kaldi and Vosk prove to be comparable, if not better, than the commercial solutions requiring a Web-API. Considering these facts, Vosk and Kaldi both appear to be valid candidates. Kaldi does offer better performance in the Librespeech benchmark but requires more fine-tuning and is generally more suited for research that solely relates to ASR [2]. Vosk additionally offers language support for Java, so the ASR solution chosen for this project is Vosk.

CodeFromVoice uses two different models provided by Vosk, both of these are RNNME models, and they mainly differ in size and memory requirement [26]. The downside is that Vosk does not offer native support for LSTM models that would improve accuracy, though it would take more processing time. RNNME models are, however, specifically for language and voice processing see Section 3.1.3 [27].

4.1.2 NLP

Arguably, the most crucial part of the system is its NLP module developed by Stanford NLP Group [3]. The NLP module's reliability and extended functionality is essential to developing scalable and robust code generation. In order to achieve this goal, the Stanford NLP Group developed a framework for statistical NLP named CoreNLP that is capable of generating logical representations from natural language text. The framework uses a variety of statistical methods and algorithms to reach its end goal, most prominently PCFG as described in Section 3.4.1.

The primary purpose of the NLP framework is to create a pipeline for natural lan-

guage processing tasks. The framework employs a classifier that trains using morphological analysis. This training process automatically extracts sentences from training documents to split words and combine strings into phrases using landmarks. The trained classifier identifies syntactic structures, such as sentences, nouns, articles, and adjectives. The results are then combined into a reduced dimensionality array to perform sentence decomposition. The other process of sentence decomposition consists of extracting sentences based on the sentence node in question and collapsing them into several phrases. This process results in a set of sentences, or an array, with intermediate logical representations that can serve as input for later code generation processes.

4.1.3 Punctuation Restoration

Combining word, sentence, and paragraph-level punctuation is essential in retaining an accurate sentence interpretation. Without punctuation, it is often difficult to determine a sentence's meaning because it provides important clues about the structure of a sentence and the meaning of individual words. A lack of punctuation can lead to invocations, as the following example: "I saw the man with binoculars," meaning that the speaker saw a man carrying binoculars. However, when punctuated, "I saw the man, with binoculars," it can be interpreted as meaning that the speaker saw a man using binoculars.

For computer interpretations, a lack of punctuation and capitalization would arise for the natural language processing algorithms such as NER, POS, and Semantic Parsing [28], which would cause any logical meaning of a sentence also to be misinterpreted, which would lead to a cascade failure for the system.

Since ASR does not have the task of predicting punctuation and capitalization's [29], a subsystem would have to be introduced that acts as a middle-way between the natural language processing of the transcription. In this system, fastPunct (`https://github.com/notAI-tech/fastPunct`) is used, a punctuating model that has been trained on a multi-task mixture of unsupervised and supervised tasks [30]. The downside of the punctuator is that it is a small model, so it introduced a problem where it could add, modify, or remove certain words. A correction is implemented for this so that it does not happen by adding an additional step of checking for character additions and removals.

4.2 System Implementation

4.3 System Pipeline

The following sections describes the process of generating code in detail. The explanation in this section should provide a frame of reference that can help conceptualize the different subsystems and their role in the whole system. Generating code starts with transcribing the input speech to text using the ASR system, followed by punctuation. After this, the text is annotated, and a constituency tree is generated using the NLP system. These annotations and the constituency tree are recursively parsed into logical components using the Logic Parsing system. During the parsing process, named entities are matched and extracted by the NER system. The resulting terminal symbols are passed to the Code translator and can then be directly translated into the code of the specified language. The pipeline model is a useful abstraction that helps explain the entire code generation process. Shown in Figure 4.2 is the pipeline model for CodeFromSpeech.

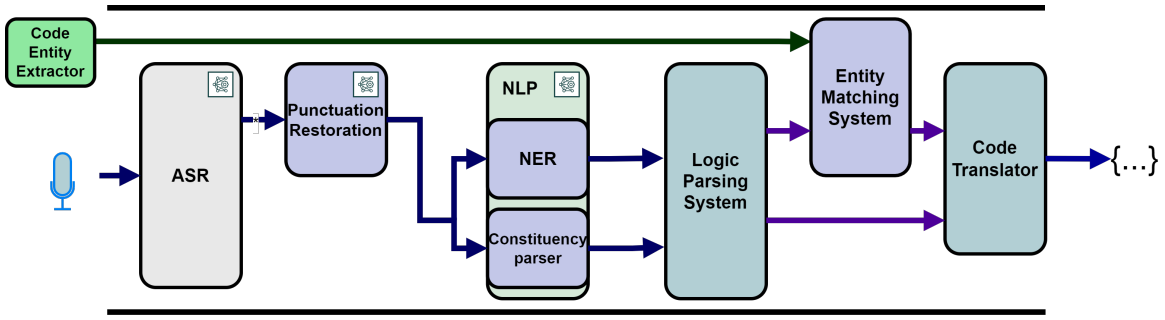


Figure 4.2: System pipeline model.

4.3.1 NLP Annotators

The CoreNLP library uses a set of properties called annotators to indicate which tools should be used. Only the annotators needed to perform a constituency parse on the given sentences for the system's requirements are used. Which limits the annotators to those that tokenize (tokenize), split the sentences (ssplit), assign part-of-speech tags (pos), lemmatize (lemma), assign named entity tags (ner), and perform regex matching (regexner). The execution time is optimized by excluding other annotators, such as those for coreference resolution and sentiment analysis.

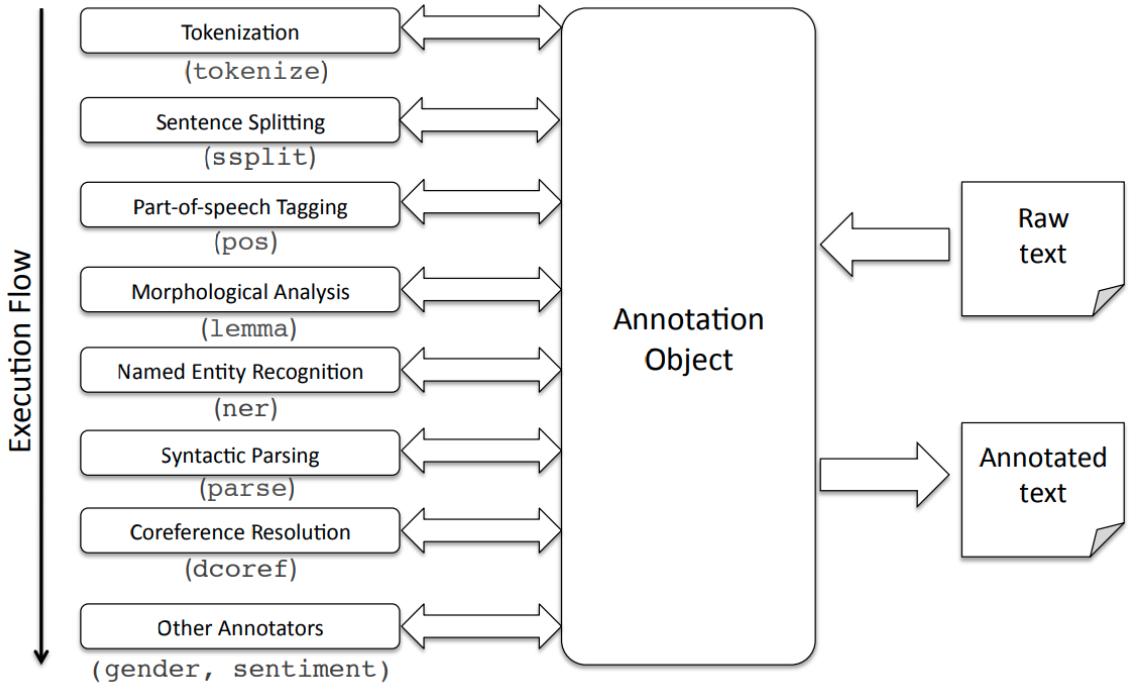


Figure 4.3: Execution flow of CoreNLP

CoreNLP allows the selection of annotator models for all of the previously mentioned annotators. For this project, the most crucial models are the POS and parse models since these are the basis for constituency parsing. The regexner annotator uses a mappings file instead of a model, as seen in Section 4.3.2. The *parse* model used by CodeFromVoice is the *IterativeCKYPCFGParser* which is as the name implies, a PCFG parser using CYK and iterative deepening see Section 3.4.1. The CYK algorithm used reduces the time complexity to $\mathcal{O}(n^3)$ which is a relatively low time complexity considering that the worst-case is asymptotic complexity [31]. Iterative deepening allows the pruning of unnecessary edges during parsing, resulting in a 60% reduction in edge comparisons for Penn Treebank 2 [31, 32].

The pos model used is the *english-bidirectional-distsim.tagger* which is a bidirectional distributional similarity tagger [33, 34]. Meaning that part-of-speech information is for a given token is extracted from its context rather than from the word itself [34]. The bidirectional part implies that relations between token are symmetric. For example: if this relationship "*The* $\overset{DET}{\Longrightarrow}$ *word*" exist for "*The*" it would imply

that the reverse relation "*The* $\underset{DET}{\Longleftarrow}$ *word*" exists for "*word*". The accuracy that the *english-bidirectional-distsim.tagger* can achieve is 97.3% token accuracy on the *WSJ 19–21 development set* [34].

4.3.2 NER System

The Named Entity Recognition system incorporates two parts. The first part is the Code Entity Extractor (CEE) system, and the second part is the Entity Matching (EM) system. The CEE system inputs the existing code in the specific context as input. This code has to be annotated according to the annotation guidelines shown in Appendix A.3. The information collected by the CEE system is compiled and serialized into one version, which is passed as mappings to the CoreNLP regexNER system [3], and the other is passed to the EM system. The compiled regexNER mappings are a list of words along with a NER tag containing information on what kind of entity it is, such as "METHOD," "CLASS," "FIELD," and modifiers (retaining access, inversion, and selection). Since the information in the regexNER mappings is insufficient to match the references with their original entities, the information sent to the CE system retains all extracted information initially. Because the regexNER mappings are prerequisites for the CoreNLP pipeline, the CEE system executes during startup or on reloading the CoreNLP pipeline. These regexNER mappings allow CoreNLP to annotate matching words with their respective NER tag.

4.3.3 Logic Parsing System

The Logic Parsing System (LPS) handles the final steps of translating speech to control logic. The LPS transforms the generated constituency tree along with the NER annotations. The prerequisite steps for the LPS are the NER extraction and annotation and the constituency parsing.

The composition of this system follows the design of a recursive descent parser. However, it does apply other non-traditional methods of parsing since constituency grammar does not constitute a Context-Free Grammar (CFG) [19]. This means that the production rules for the parser can depend on preceding and future subsequent production rules. If a production rule shares a two-way dependency with another, then the operation that is last in the execution contains a hook that reverts the parsing before the first rule, only containing the context specified in the hook. The difference between the LPS and a traditional parser is the use of these two-way dependent production rules [19]. The downside of this approach is that the time complexity of determining if an input can produce an output scales with $\mathcal{O}(2^n)$ rather than $\mathcal{O}(n^3)$

for exhaustive search [35]. Though not all rules behave in such a way, rules that are designated to be ”default” or ”canonical” cannot contain a two-way dependency, and these rules are formally defined in Section 5.1.

The canonical production rules are categorized into five categories based on the relation to constituency [36] and production target:

1. Sentence subdivision rules - RS
2. Clause separation rules - RC
3. Phrase extraction rules - RP
4. Entity extraction rules - RE
5. Augmentation rules - RA

4.3.4 Production Rules

The rules shown in Table 5.1 were created by learning the linguistic meaning of the constituent tags and relating those to an expected input pattern, e.g., ”When X do Y.”, ”Do Y when X.”. After this, an analysis of what patterns PTB tags exhibit was required. For the analysis, the testing dataset (not the same as the evaluation dataset) was used. The most apparent of these patterns are those relating to clause structure, which is the basis for the clause separation rules. The exact design of the clause rules are refined based on the paper ”*Natural Language Parsing as Statistical Pattern Recognition*” [36]. The role that clause separation has is to separate execution specifiers detailing the condition or scope and operations detailing the action.

The next step after clause separation is to identify connected groups of phrases that either correspond to one entity or one expression e.g. ”RPM value of the motor” could correspond to either *Motor.getRPM();* or *Motor.setRPM(val);*. To detect these connected groups, the constituency inheritance is reliable with some exceptions shown in Table 5.1 ID: RP1. [36]. The entity matching system is used for cases where the inheritance are not particularly useful. This type of grouping is the basis for the Phrase extraction rules.

The entity extraction rules and augmentation rules are responsible for pairing entity references and altering the parsing sequence. Since the phrase and clause rules were developed first, they could be applied, reducing the number of possible interpretations.

The development of these rules was iteratively implemented and tested based on a set of target outputs. Methods in "*The Theory of Parsing, Translation, and Compiling*" *chapter 4.1.3* is used as reference [37]. The formal definition of the LPS grammar is shown in the equation below with the same format as in Section 3.2.2.

$$G_{LP} = \langle \{V_e, V_c, V_n\}, T, S_r, \{\{RS_c, RC_c, RP_c, RE_c, RA_c\}, \{RS_n, RC_n, RP_n, RE_n, RA_n\}\}\rangle$$

$\mathbf{V_e}$: Entity reference leaf nodes,
$\mathbf{V_c}$: PTB tagged Nodes,
$\mathbf{V_n}$: Leaf nodes (regular english), such that $V_n \cap V_e = \emptyset$,
T: Set of terminal symbols (see Section 4.3.6),
$\mathbf{S_r}$: Root node,
$\mathbf{RX_c}$: Set of canonical production rules with type "X" (see Table 5.1),
$\mathbf{RX_n}$: Set of non-canonical production rules with type "X", such that$RX_n \cap RX_c = \emptyset$

Figure 4.4: LPS grammar definition.

4.3.5 Production rules examples

The canonical production rules listed in Section *5.1* have two key roles, subdividing and extracting. Subdividing rules restrict the scope for a group of following rules. These rules do not produce any terminal symbols. The subdividing rules are restricted to the first four categories RS, RC, RP, and RE. The rule RC1 shown in *Figure 4.5* is an example of a subdividing rule. The produced result is the first SBAR node along with its children. In normal parsing operation, this set would be further processed by the rules RP2, RE2, RE3, and RA1.

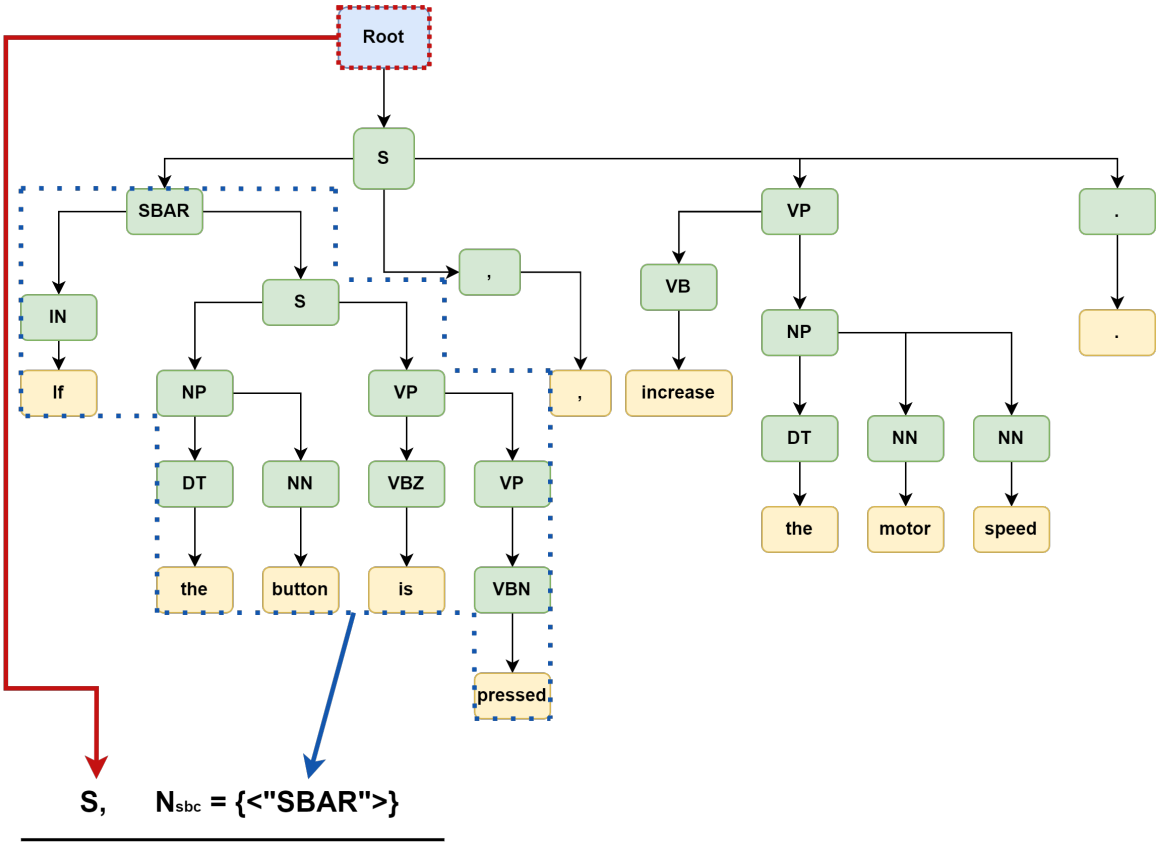


Figure 4.5: Illustration of the production rule RC1 (tags listed in A.4).

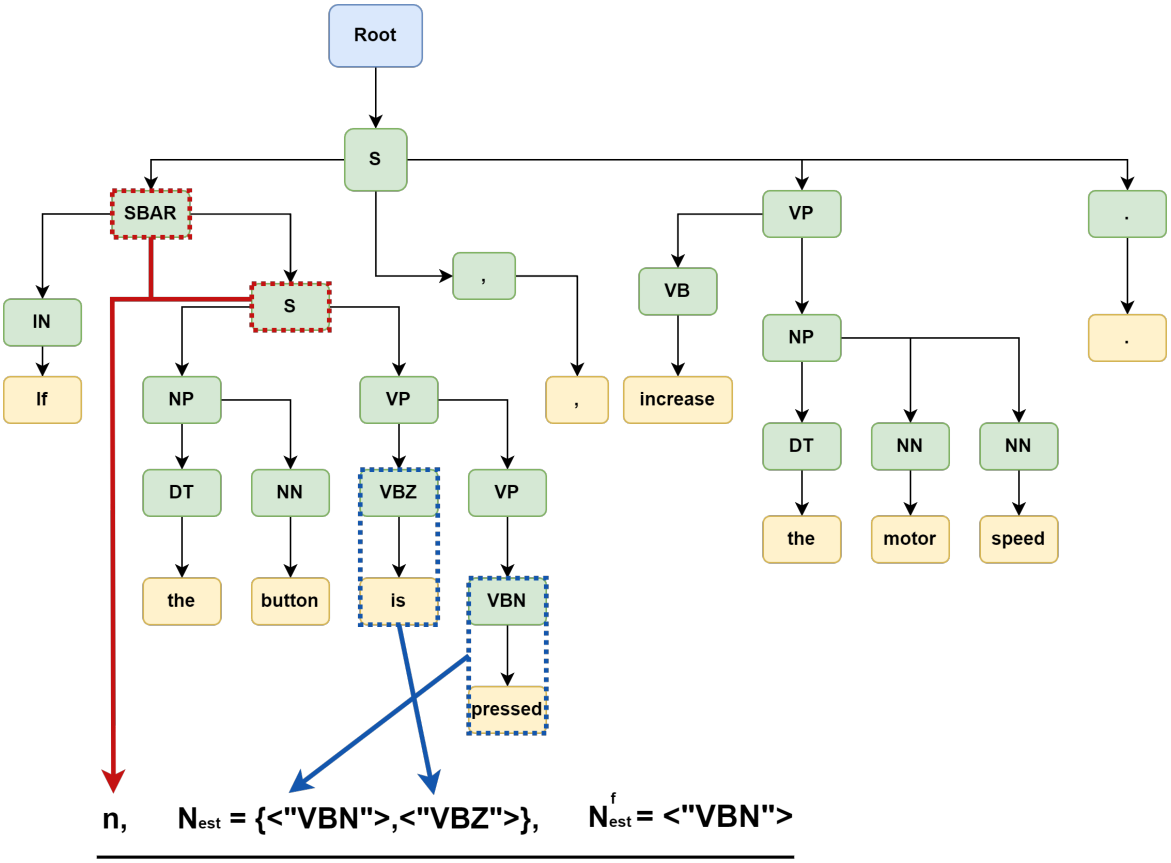


Figure 4.6: Illustration of the production rule RE2 (tags listed in A.4).

Extracting rules produce one or more nodes detailing a specific aspect of the final logical expression. The result will either be processed by a non-canonical rule, used to extract a named entity, or directly translated to a terminal symbol. The rule RE2 shown in Figure 4.6 is an example of an extracting production rule. In this case, only one node is selected following the order of precedence. In a regular parsing operation, the resulting node from this rule would be passed along with the result of rules RE3 and RA1 would form an entity tuple. This entity tuple would be passed to the NER system for exact matching.

4.3.6 Terminal and non-terminal symbols

Appendix Section A.2 describes the set of terminal symbols. These terminal symbols make up the formal language that is the basis of our code generation. Someone familiar

with compiler design would notice these terminal symbols have more compositionality than is typical for a formal language. The reason for this compositionality is to mitigate the drawback of exhaustive search caused by top-down parsing [19]. Furthermore, it reduces the number of rules needed for a complete derivation.

4.4 Evaluation

4.4.1 Existing Datasets

Evaluating the accuracy of CodeFromVoice required data that could emulate an actual use case. Initially, an established dataset was considered for this task as a time-saving measure. Three such datasets fit the general description, CoNaLa, Django, and CodeXGLUE [38–40]. There are several issues with this approach. Firstly, the type of code did not match for CoNaLa and CodeXGLUE. CoNaLa is based on StackOverflow questions. These have multiple sentences for each example [38]. The text to code data in CodeXGLUE maps one sentence to a complete function or segment, e.g., "mixed case to underscores.".

The second and probably most significant issue with using existing datasets is that CodeFromVoice is context-dependent. Meaning that functions and variables need mappings, as described in Section 4.3.2. Because the data varies, many entities would need to get constructed to use even a fraction of a dataset.

4.4.2 Evaluation Data

The evaluation data used in Section 5.2 are divided into three parts, each representing one type of terminal output. These are *If statements*, *While loops* and *Function calls*. Each test contains input data and reference data. The reference data is a string containing manually written code. The input data contains an unpunctuated sentence, a punctuated sentence, and one or multiple audio files. Each test type additionally contains classes, functions, and fields used in the tests for entity references. This structure allows the selection of subsystems to be tested. The metrics that are calculated from the tests are WER and EM see Appendix A.1. When calculating WER for code, syntactic symbols such as "{ ; . (" and line breaks are removed. A test example is displayed in Figure 4.7. A handful more are shown in Appendix A.5, which should help convey the types of sentences that get parsed.

```
Test:
    "if_statement_137"
Audio Files:
    {"if_statement_137_audio_1.wav", "if_statement_137_audio_2.wav"}
Unpunctuated Sentence :
    "turn the heater on if the button is pressed and the heater is off"
Punctuated Sentence :
    "Turn the heater on if the button is pressed and the heater is off."
Code :
    if(Button.isPressed() && !Heater.isActive())
    {
        Heater.start();
    }
```

Figure 4.7: If statement test example.

5 Result

5.1 Production Rules

5.1.1 Definition of Canonical Rules

When describing the canonical rules the notation follows standard Set Theory [41] with some exceptions shown below:

- Start/Root: $\mathbf{S}$
- Set of all symbols in $\mathbf{S}$: $\boldsymbol{N_S}$
- Set of all non-terminal in $\boldsymbol{S}$: $\boldsymbol{N_{SN}}$
- Set of all terminal in $\boldsymbol{S}$: $\boldsymbol{N_{ST}}$
- Symbol with value "$\mathbf{X}$": $\langle \boldsymbol{''X''} \rangle$
- Symbol with function tag value "$\mathbf{X}$": $\langle \boldsymbol{\$'' - X''} \rangle$
- Non-Terminal symbol: $\boldsymbol{n, c}$
- Terminal symbol: $\boldsymbol{t}$
- Parent of Non-Terminal $\boldsymbol{n, c}$: $\boldsymbol{n', c'}$
- Set of children of non-terminal symbol $\boldsymbol{n_x}$: $\boldsymbol{N_x}$
- Derivation/production from $\mathbf{s}$ to $\mathbf{N}$ with the grammar $\mathbf{G}$: $\boldsymbol{s} \underset{\boldsymbol{G}}{\Rightarrow} \boldsymbol{N}$
- First element in set : $c = \boldsymbol{N^f}$, Such that $c \neq \emptyset$

Additionally, the full list of PTB tags is listed in Appendix A.4.

Table 5.1: Definition of canonical parsing rules.

ID	Rule name	Input	Set of possible outputs	Production rule
RE1	Operator Adverb Rule	n	$N_{adv} = \{c \in N_{SN}; c \in \{\langle''RB''\rangle, \langle''RBR''\rangle, \langle''RBS''\rangle\}$	$n \underset{G}{\Rightarrow} N_{adv}$
RC1	Subordinate Clause Splitting Rule	S	$N_{sbc} = \{c \in N_{SN}; c = \langle''SBAR''\rangle\}$	$S \underset{G}{\Rightarrow} N^f_{sbc}$
RE2	Execution Specifier Target Rule	$n \in N_{SN} \wedge \langle''SBAR''\rangle \in \{n, n'\}$	$N_{est} = \{c \in N_{NS}; N_{c'} \ni c \wedge \langle''VBN''\rangle \notin N_{c'} \wedge c \subset \{\{c1 \in N_{c'}; c1 = \langle''VBN''\rangle\}, \{c2 \in N_{c'}; c2 = \langle''ADJP''\rangle\}, \{c3 \in N_{c'}; c3 = \langle''PP''\rangle\}, \{c4 \in N_{c'}; c4 = \langle''VBZ''\rangle\}\}\}$	$n \underset{G}{\Rightarrow} N^f_{est}$
RE3	Noun NER Rule	n	$N_{nnr} = \{c \in N_{SN};\ c \notin N_{ST} \wedge c \in \{\langle''NN''\rangle, \langle''NNS''\rangle, \langle''NNP''\rangle, \langle''NNPS''\rangle\}\}$	$n \underset{G}{\Rightarrow} N_{nnr}$
RP1	Repeating Execution Specifier Rule	n	$N_{fes} = \{c \in N_{SN}; \langle''CD''\rangle \in N_{c'} \wedge (\langle''NP''\rangle \in N_{c'} \vee (N_{c'} \cap \{\langle\$\text{''-TMP''}\rangle, \langle\$\text{''-EXT''}\rangle\} \neq \emptyset)) \wedge \langle''NNS''\rangle \in N_{c''}\}$	$n \underset{G}{\Rightarrow} N^f_{fes}$
RP2	Noun Refactoring Rule	n	$N_{npr} = \{c \in N_{SN}; c = \langle''NP''\rangle\}$	$n \underset{G}{\Rightarrow} N^f_{npr}$
RE4	Operation Entity Rule	$n \in N_{n'} \wedge n' = \langle''VP''\rangle$	$N_{oe1} = \{c \in N_{oer}; c1 \in N_{c'} \wedge c1 \in \{\langle''SBAR''\rangle, \langle''SINV''\rangle, \langle''SBARQ''\rangle\} \wedge \langle''DT''\rangle \in N_{c'}\}$ $N_{oe2} = \{c2 \in N_{oe2}; c1 \in N_{oe1} \wedge c2 \in N_{c1} \wedge c2 = \langle''NP''\rangle\}$	$n \underset{G}{\Rightarrow} N_{oe2}$
RE5	Operation Target Rule	n	$N_{ot} = \{c \in N_{SN}; c \in \{\langle''VB''\rangle, \langle''VBP''\rangle\}$	$n \underset{G}{\Rightarrow} N_{ot}$
RP3	Preposition Classifier Rule	n	$N_{pr} = \{c \in N_{SN}; c' = \langle''PP''\rangle \wedge c = \langle''IN''\rangle\}$	$n \underset{G}{\Rightarrow} N_{pr}$
RP4	Compound Quantifier Rule	n	$N_{qp} = \{c \in N_{SN}; c = \langle''QP''\rangle\}$	$n \underset{G}{\Rightarrow} N_{qp}$
RA1	Verb Modifier Rule	n	$N_{vm} = \{c \in N_{SN}; c'' = \langle''VP''\rangle \wedge ((c \in \{\langle''VP''\rangle \wedge c' = \langle''PP) \vee (c = \langle''RP''\rangle \wedge c' = \langle''PRT''\rangle) \vee (c = \langle''RB''\rangle \wedge c' = \langle''ADVP''\rangle))\}$	$n \underset{G}{\Rightarrow} N_{vm}$
RP4	Verb Phrase Extraction Rule	n	$N_{vp} = \{c \in N_{SN}; c = \langle''VP''\rangle\}$	$n \underset{G}{\Rightarrow} N_{vp}$

5.2 Evaluation Results

The evaluation results are separated into three sections for different system parts. Section 5.2.1 tests the NLP, LPS and NER system. Section 5.2.2 tests all the previous with the addition of the punctuator. Section 5.2.3 tests all systems.

5.2.1 Parsing Evaluation Results

The evaluation of the LPS system and, in extension, the production rules listed in Section 5.1.1 is separated into three tests containing one type of Logical component. The results of these tests are shown in Table 5.2. For these tests, the input text is manually punctuated.

Table 5.2: Evaluation results of Logic Parsing system.

Metric	*If statement test*	*While loop test*	*Function call test*
Code Exact Match (%)	73.07	71.42	83.32
Code WER (%)	18.52	12.64	12.77
Number of Tests	156	98	111
Total Computation Time (s)	17.24	12.33	13.11

5.2.2 Punctuator parsing result

To view the how the impact on accuracy of the punctuator the test shown in Table 5.3 is given without any punctuation or capitalization.

Table 5.3: Evaluation results of Logic Parsing system and punctuator.

Metric	*If statement test*	*While loop test*	*Function call test*
Code Exact Match (%)	71.04	71.42	78.37
Code WER (%)	18.92	13.30	13.21
Number of Tests	156	98	111
Punctuator Exact Match (%)	98.07	85.71	86.48
Total Computation Time (s)	32.90	21.29	24.14

5.2.3 Full System Parsing Result

The full system's accuracy is the most important since it is how this system is designed to be used. The full system accuracy shown in Table 5.4 is produced from audio recordings and averaged for two different speakers using different microphones. The audio files are recorded at 32kHz sampling rate with 2 bytes/sample and downsampled to 16kHz. The reasoning for downsampling is to reduce the impact of the type of microphone used and speed up the processing time.

Table 5.4: Evaluation results of Logic Parsing system, punctuator and ASR.

Metric	*If statement test*	*While loop test*	*Function call test*
Code Exact Match (%)	67.30	66.32	72.97
Code WER (%)	19.53	19.81	15.51
Number of Tests	156	98	111
Punctuator Exact Match (%)	62.17	66.32	74.77
ASR Exact Match (%)	68.58	54.05	81.08
ASR WER (%)	5.32	6.52	4.12
Total Computation Time (s)	34.26	21.93	24.89

6 Conclusion and Discussion

The system presented in this thesis shows semantic parsing and code generation from natural language .

6.1 Discussion

The system presented in this thesis shows one approach to semantic parsing and code generation from natural language. The goal is for the system to reach a WER < 0.25, and the system achieves this in all tests, as shown in Section 5.2.1, 5.2.2, and 5.2.3. Though the system achieves this goal, much work remains regarding the capability to allow input with more compositionality. It should also be noted that the task of semantic parsing does not have a tightly defined evaluation metric. An evaluation metric that could encapsulate both the complexity of the logic and the accuracy would give a much clearer picture of the semantic parsing capabilities. This system shows that an intermediate syntactic parsing of natural language such as constituency trees is sufficient for simple context-specific semantic parsing using traditional methods and tools. Considerable work is still required to achieve human-level accuracy, which is not surprising, but the results are significant. The system produces proper control flow for if statements, while loops and functions call with more than 0.7 exact match probability.

Because the ASR punctuator significantly impacts the total system accuracy and NLP systems, some inaccuracies are inevitable. It was tough to gauge how greatly these subsystems would impact during this project. One such example is the punctuator which was not considered an impactful source of inaccuracies, but the evaluation showed a significant impact. The evaluation can be found in Section 5.2.2. This is most likely due to the high reliance on the accurate placement of commas that the constituency parser requires to produce consistent results. The LPS does have some measures of repunctuation, but these do not cover all cases and are primarily to contract repeating problems with the punctuator.

Though the results that system achieves align with the goal set out, the way of parsing

the semantics did change multiple times. The method of semantic parsing presented in Section 4.2 was established after several other methods proved fruitless, such as Universal Dependency parsing and Lambda calculus extraction utilizing the UDepLambda semantic parsing system [4]. Another approach that was explored is to train a neural network model for semantic parsing abstract syntax trees, based on similar methods to TranX [10]. The final shift to the more traditional method of parsing presented in Section 4.3.3 was after two-thirds of the set time limit. The exploration of other these methods did give insight and understanding into the field of Computational Linguistics. This, however, allowed for less fine-tuning and testing of the system.

6.1.1 Economical aspects

A system that automatically generates control logic from ordinary speech has considerable economic benefits. Such as reducing the need for human programming and lowering the cost of production. Since the generated output could be produced, corrected, and tested faster than a human could, the system would not have to be comparable to a human programmer to be useful. Furthermore, this could allow workers specializing in non-technological fields to utilize their knowledge of a given task in a computer program. For tasks that involve specificity or complexity beyond this system's capabilities, it could instead be used to transcribe or interpret instructions into pseudocode. This pseudocode could then be used to clear up or remove ambiguities in the instructions that a programmer would receive [42].

6.1.2 Ethical Aspects

The ethical aspects of many AI projects often boil down to if it is classified as a "Human-Enhancing Innovation" (HEI) or a "Human-Replacing Innovation" (HRI) [42]. Considering that the goal of this project is to generate code from speech, human input is still needed. So this is not meant to replace humans but enhance the user's capabilities. Though this distinction lacks nuance, enhancing and replacing humans occur to different degrees. For instance, the introduction of the email was shown to strongly correlate with the reduction of mail sent through the postal system, reducing the number of postal workers [43]. However, the shift to email is almost universally viewed as an enhancing technology in this case. This is most likely because sorting and delivering mail as a job is viewed differently from most white-collar or creative work types. The value or consideration of a project's work mainly determines if it is an enhancement or replacement.

6.1.3 Risks

The risks introduced by this system (besides the ones mentioned in Section 6.1.2) are entirely based on where the generated code is used. If this system is to be further developed, it could be released. It should be made abundantly clear to the user that no safety-critical operations should be done based on this system's prediction. Additionally, code generated using CodeFromVoice is still the responsibility of the user. Listing examples of dangers from incorrect code is left as an exercise for the reader.

6.1.4 Scientific Method

One aspect of the result section that may seem lacking is presenting conclusive evidence based on established methods. The results presented in Section 5 are instead based on evaluation methods and data created for this specific project. This might seem to disregard the scientific method invalidating the claims made in Section 6 [44]. However, the underlying reason for creating this system is not to prove that the methods result in comparative advantage in an established benchmark such as CONALA or CodeXGLUE [38, 40]. The purpose is to design and develop a good and effective system in the targeted use-cases. Even though this kind of critique is somewhat justified, the measurements shown in Tables 5.2, 5.3, 5.4 is gathered with scientific rigor [44]. All measurements are gathered in a well-controlled and unbiased manner, meaning that no uncontrolled factors impacted the result. The produced evaluation data is not cherry-picked to produce optimal results. The interpretation and reporting of these results reflect the results with the goal set out.

6.2 Conclusion

Developing a system that can generate control logic in the form of code from speech with a WER of less than 25% is successful. This system is named CodeFromVoice and is composed of four main subsystems, The ASR, NLP, NER, and Logic Parsing systems. Generating code starts with transcribing the input speech to text using the ASR system and applying punctuation. After this, the text is annotated, and a constituency tree is generated using the NLP system. These annotations and the constituency tree are recursively parsed into logical components using the Logic Parsing system. During the parsing process, named entities are matched and extracted by the NER system. The resulting terminal symbols can then be directly translated into the code of the specified language. The fact that is most apparent following this project is the sheer scope and complexity of problems semantic parsing introduces. Problems

that impact the Logic parsing system design, selection of tools, and the method of integrating the other systems. However, CodeFromVoice shows surprisingly good results and is able to generate conditional statements, loops, and function calls which are the fundamental control structures of code. These accomplishments are due to excellent tools such as CoreNLP and Vosk and the NER and Logic Parsing system.

Bibliography

[1] Le THM, Chen H, Babar MA. Deep learning for source code modeling and generation: Models, applications, and challenges. ACM Comput Surv. 2020;53(3):1-38.

[2] VOSK offline speech recognition API;. Accessed: 2022-3-3. https://alphacephei.com/vosk/index.

[3] Manning C, Surdeanu M, Bauer J, Finkel J, Bethard SJ, McClosky D. The Stanford CoreNLP Natural Language Processing Toolkit. In: Association for Computational Linguistics (ACL) System Demonstrations; 2014. p. 55-60.

[4] Reddy S, Täckström O, Petrov S, Steedman M, Lapata M. Universal Semantic Parsing. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing. Stroudsburg, PA, USA: Association for Computational Linguistics; 2017. .

[5] Rakotomalala F, Randriatsarafara HN, Hajalalaina AR, Ravonimanantsoa NMV. Voice User Interface: Literature review, challenges and future directions. SYSTEM THEORY, CONTROL AND COMPUTING JOURNAL. 2021 Dec;1(2):65–89. Available from: http://stccj.ucv.ro/index.php/stccj/article/view/26.

[6] Lopatovska I, Rink K, Knight I, Raines K, Cosenza K, Williams H, et al. Talk to me: Exploring user interactions with the Amazon Alexa. Journal of Librarianship and Information Science. 2019 Dec;51(4):984–997. Available from: http://journals.sagepub.com/doi/10.1177/0961000618759414.

[7] Braun D, Hernandez-Mendez A, Matthes F, Langen M. Evaluating Natural Language Understanding Services for Conversational Question Answering Systems. In: Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue. Saarbrücken, Germany: Association for Computational Linguistics; 2017. p. 174–185. Available from: http://aclweb.org/anthology/W17-5522.

[8] Chomsky N. Three models for the description of language. IRE Trans Inf Theory. 1956;2:113-24.

[9] Yin P, Neubig G. A Syntactic Neural Model for General-Purpose Code Generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Vancouver, Canada: Association for Computational Linguistics; 2017. p. 440–450. Available from: http://aclweb.org/anthology/P17-1041.

[10] Yin P, Neubig G. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Stroudsburg, PA, USA: Association for Computational Linguistics; 2018. .

[11] Dale R. GPT-3: What's it good for? Natural Language Engineering. 2021;27(1):113–118.

[12] Zaremba W, Brockman G, OpenAI. About OpenAI Codex,. OpenAI; 2021. https://openai.com/blog/openai-codex/. Available from: https://openai.com/blog/openai-codex/.

[13] Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, et al. Evaluating Large Language Models Trained on Code. CoRR. 2021;abs/2107.03374. Available from: https://arxiv.org/abs/2107.03374.

[14] Bishop CM. Neural networks and their applications. Review of Scientific Instruments. 1994;65(6):1803-32. Available from: https://doi.org/10.1063/1.1144830.

[15] Shalev-Shwartz S, Ben-David S. Understanding machine learning: From theory to algorithms. Cambridge, England: Cambridge University Press; 2014.

[16] Sherstinsky A. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. Physica D. 2020;404(132306):132306.

[17] Mikolov T, Kombrink # S, Deoras A, Burget L, Honza J, Černocký. RNNLM -recurrent neural network language modeling toolkit;. Accessed: 2022-5-9. http://www.fit.vutbr.cz/~imikolov/rnnlm/rnnlm-demo.pdf.

[18] Brown PF, Della Pietra VJ, Desouza PV, Lai JC, Mercer RL. Class-based n-gram models of natural language. Computational linguistics. 1992;18(4):467-80.

[19] Linz P. An Introduction to Formal Languages and Automata. 6th ed. Jones Bartlett Learning; 2016.

[20] Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge, UK : New York: Cambridge University Press; 1998.

[21] Clarke J, Goldwasser D, Chang MW, Roth D. Driving semantic parsing from the world's response;. Accessed: 2022-4-30. https://aclanthology.org/W10-2903.pdf.

[22] Panayotov V, Chen G, Povey D, Khudanpur S. Librispeech: An ASR corpus based on public domain audio books. In: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE; 2015. .

[23] Rousseau A, Deléglise P, Estève Y. TED-LIUM: an automatic speech recognition dedicated corpus. In: In Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12; 2012. .

[24] Kenarsari A. Speech-to-Text Benchmark; 2022. Accessed: 2022-2-26. https://github.com/Picovoice/speech-to-text-benchmark.

[25] Kaldi ASR;. Accessed: 2022-2-26. https://kaldi-asr.org/models/m13.

[26] Models;. Accessed: 2022-2-26. https://alphacephei.com/vosk/models.

[27] Chen X, Liu X, Qian Y, Gales MJ, Woodland PC. CUED-RNNLM—An open-source toolkit for efficient training and evaluation of recurrent neural network language models. In: 2016 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE; 2016. p. 6000-4.

[28] Nguyen B, Nguyen VBH, Nguyen H, Phuong PN, Nguyen T, Do QT, et al. Fast and Accurate Capitalization and Punctuation for Automatic Speech Recognition Using Transformer and Chunk Merging. CoRR. 2019;abs/1908.02404. Available from: http://arxiv.org/abs/1908.02404.

[29] Zelasko P, Szymanski P, Mizgajski J, Szymczak A, Carmiel Y, Dehak N. Punctuation Prediction Model for Conversational Speech. CoRR. 2018;abs/1807.00543. Available from: http://arxiv.org/abs/1807.00543.

[30] Roberts A, Raffel C. Exploring transfer learning with T5: The text-to-text transfer transformer; 2020. Available from: https://ai.googleblog.com/2020/02/exploring-transfer-learning-with-t5.html.

[31] Manning C. Javadoc IterativeCKYPCFGParser;. Accessed: 2022-5-26. https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/parser/lexparser/IterativeCKYPCFGParser.html.

[32] Tsuruoka Y, Tsujii J. Iterative CKY Parsing for Probabilistic Context-Free Grammars. vol. 3248; 2004. p. 52-60.

[33] Kristina Toutanova JSARMGCMJB Miler Lee. Javadoc MaxentTagger;. Available from: https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/tagger/maxent/MaxentTagger.html.

[34] Manning C. Part-of-Speech Tagging from 97% to 100%: Is It Time for Some Linguistics? In: Gelbukh AF, editor. Computational Linguistics and Intelligent Text Processing. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011. p. 171-89.

[35] JSL volume 37 issue 2 Cover and Front matter and Errata. Journal of Symbolic Logic. 1972;37(2):f1–f7.

[36] Magerman DM. Natural Language Parsing as Statistical Pattern Recognition. CoRR. 1994;abs/cmp-lg/9405009. Available from: http://arxiv.org/abs/cmp-lg/9405009.

[37] Aho AV, Ullman JD. The Theory of Parsing, Translation, and Compiling. USA: Prentice-Hall, Inc.; 1972.

[38] Yin P, Deng B, Chen E, Vasilescu B, Neubig G. Learning to mine aligned code and natural language pairs from stack overflow. In: Proceedings of the 15th International Conference on Mining Software Repositories. New York, NY, USA: ACM; 2018. .

[39] Oda Y, Fudaba H, Neubig G, Hata H, Sakti S, Toda T, et al. Learning to generate pseudo-code from source code using statistical machine translation. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2015. .

[40] Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. 2021.

[41] Bagaria J. Set Theory. In: Zalta EN, editor. The Stanford Encyclopedia of Philosophy. Winter 2021 ed. Metaphysics Research Lab, Stanford University; 2021. .

[42] Trajtenberg M. Nber working paper series Ai as the next gpt: A political-economy perspective; 2018. Accessed: 2022-4-27. https://www.nber.org/system/files/working_papers/w24245/w24245.pdf.

[43] Computer Communications. 1980 Apr;3(2):91. Available from: https://linkinghub.elsevier.com/retrieve/pii/0140366480901127.

[44] Cohen MF. An Introduction to Logic and Scientific Method. Read Books Ltd; 2011. Google-Books-ID: 5Ep8CgAAQBAJ.

[45] Klakow D, Peters J. Testing the correlation of word error rate and perplexity. Speech Communication. 2002 Sep;38(1–2):19–28. Available from: https://linkinghub.elsevier.com/retrieve/pii/S0167639301000413.

[46] Xia F. The Part-Of-Speech Tagging Guidelines for the Penn Chinese Treebank (3.0). 2000 11.

A Appendix

A.1 WER Definition

Word Error Rate (WER) is the primary metric used in this report. WER is used to calculate the number rate of substitutions, deletions, and insertions of a given sequence of words or tokens [45]. As shown in Equation A.1.

$$WER = \frac{S + D + I}{N} \tag{A.1}$$

Where **N**: number of words, **S**: substitutions, **D**: deletions, **I**: insertions

A.2 List of Terminal Symbols

1. Function with arguments
2. Function no arguments
3. Function with arguments, classless
4. Function no arguments, classless
5. Field
6. Field, classless
7. If Statement
8. If Else Statement
9. While Loop
10. For Loop
11. Foreach Loop
12. Binary Value

13. Binary Operator
14. Numeric value
15. Numeric Comparison
16. Numeric Operator
17. Function Grouping

A.3 Annotation Guidelines

The following code listings show the annotation guidelines required by the CEE system.

Listing A.1: Annotation Guidelines: Classes

```
/**
 * Each name following "@nerref" corresponds to word that is used to
    reference this class
 * separated by "/"
 * The class needs to be mentioned in connection with a field or term if
    those are to be extracted
 * @nerref example/example class
 */
public class ExampleClass
{
...
}
```

Listing A.2: Annotation Guidelines: Fields

```
/**
 * @nerref example/example class
 */
public class ExampleClass
{
 /**
    * Same format for fields/variables with the addition of "GLOBAL"
    * reference words appended with "{GLOBAL}" does not require the
       mentioning of the class to be accessed
    * !note that global entities have lower precedence than if a class was
       mentioned
    * @nerref value/number/example constant{GLOBAL}
    */
   public static int INT_VALUE;

   /**
    * Private functions or fields won't be considered
    */
   private static boolean isActive = false;
   ...
}
```

Listing A.3: Annotation Guidelines: Functions

```
/**
 * @nerref example/example class
 */
public class ExampleClass
{
    ...
    /**
     * Same format for functions with the addition of conditional method
        selection
     * by adding a modifier that correspond to a boolean value either the
        stop()
     * or start function will be called. E.g. "...set example to on..." ->
        ExampleClass.start();
     * (functions can also utilize "{GLOBAL}")
     * @nerref start/initiate/turn{BOOL_COND,TRUE}/set{BOOL_COND,TRUE}
     */
    public static void start()
    {...}

    /**
     * "...set example to off..." -> ExampleClass.stop();
     * @nerref
        stop/deactivate/close/turn{BOOL_COND,TRUE}/set{BOOL_COND,TRUE}
     */
    public static void stop()
    { }
    ...
}
```

Listing A.4: Annotation Guidelines: Function inversion

```
/**
 * @nerref example/example class
 */
public class ExampleClass
{
    ...
    /**
     * Function types relating to conditional logic can be inverted based
        reference
     * "... example is on..." ->ExampleClass.isOn(), "...example is
        off..."->!ExampleClass.isOn()
     * same goes for turned but is another way of writing it.
     * @nerref is on{RETURN,NORMAL}/is
        off{RETURN,INVERTED}/turned{RETURN,[on:NORMAL],[off:INVERTED]}
     */
    public static boolean isOn()
    {
           return isActive;
    }
}
```

A.4 Penn Treebank Tagset

The following tables contain the Penn TreeBank(PTB) tagset for clause level, phrase level, and word level. Function tags such as "*-TMP*","*-EXT*" are ommited from this list but can be found in the article "*The Part-Of-Speech Tagging Guidelines for the Penn Chinese Treebank (3.0)*" [46].

Table A.1: Penn treebank clause tags [46]

Description	**Tag**
Simple declarative clause	*S*
Subordinate clause	*SBAR*
Direct question	*SBARQ*
Inverted declarative sentence	*SINV*
Inverted yes/no question	*SQ*

Table A.2: Penn treebank phrase tags [46].

Description	**Tag**
Adjective phrase	*ADJP*
Adverb phrase	*ADVP*
Conjunction phrase	*CONJP*
Fragment	*FRAG*
Interjection	*INTJ*
List marker	*LST*
Not a constituent	*NAC*
Noun phrase	*NP*
Complex noun phrase	*NX*
Prepositional phrase	*PP*
Parenthetical	*PRN*
Particle	*PRT*
Quantifier phrase	*QP*
Reduced relative clause	*RRC*
Unlike coordinated phrase	*UCP*
Verb phrase	*VP*
Wh adjective phrase	*WHADJP*
Wh adverb phrase	*WHAVP*
Wh noun phrase	*WHNP*
Wh prepositional phrase	*WHPP*
Unknown	*X*

Table A.3: Penn treebank POS tags [46].

Description	**Tag**
Coordinating conjunction	*CC*
Cardinal number	*CD*
Determiner	*DT*
Existential there	*EX*
Foreign word	*FW*
Preposition or subordinating conjunction	*IN*
Adjective	*JJ*
Adjective comparative	*JJR*
Adjective superlative	*JJS*
List item marker	*LS*
Modal	*MD*
Noun singular	*NN*
Noun plural	*NNS*
Proper noun singular	*NNP*
Proper noun plural	*NNPS*
Predeterminer	*PDT*
Possessive ending	*POS*
Personal pronoun	*PRP*
Possessive pronoun	*PRP$*
Adverb	*RB*
Adverb comparative	*RBR*
Adverb superlative	*RBS*
Particle	*RP*
Symbol	*SYM*
To	*TO*
Interjection	*UH*
Verb base form	*VB*
Verb past tense	*VBD*
Verb gerund or present participle	*VBG*
Verb past participle	*VBN*
Verb non 3rd person singular present	*VBP*
Verb 3rd person singular present	*VBZ*
Wh determiner	*WDT*
Wh pronoun	*WP*
Possessive wh pronoun	*WP$*
Wh adverb	*WRB*
.	.
,	,

A.5 Evaluation Data Examples

The regexNER mappings used in the evaluation dataset are shown in Table A.4.

```
Test:
    "while_loop_86"
Audio Files:
    {"while_loop_86_audio_1.wav", "while_loop_86_audio_2.wav"}
Unpunctuated Sentence :
    "log it while the rpm value is above the motor threshold"
Punctuated Sentence :
    "Log it while the RPM value is above the motor threshold."
Code :
    while( Motor.getSpeed() > Motor.getThreshold() )
    {
        GlobalFunctions.log();
    }
```

Figure A.1: While loop test 86.

```
Test:
    "if_statement_134"
Audio Files:
    {"if_statement_134_audio_1.wav", "if_statement_134_audio_2.wav"}
Unpunctuated Sentence :
    "when sensor one has triggered more times than sensor two save the
        temperature and turn the motor off"
Punctuated Sentence :
    "When sensor one has triggered more times than sensor two, save the
        temperature and turn the motor off."
Code :
    if(Sensor.one.timesTriggered > Sensor.two.timesTriggered)
    {
        GlobalFunctions.save(TempSensor.getTemp());
        Motor.stop();
    }
```

Figure A.2: If statement test 134.

```
Test:
    "function_calls_31"
Audio Files:
    {"function_calls_31_audio_2.wav", "function_calls_31_audio_2.wav"}
Unpunctuated Sentence :
    "run function one and deactivate motor one"
Punctuated Sentence :
    "Run function one and deactivate motor one."
Code :
    GlobalFunctions.func1();
    Motor.primary.stop();
```

Figure A.3: Function calls test 31.

Table A.4: RegexNer mappings used in tests.

Regex Entity reference	**Type** (MWT_* : Multi Word Token)
momentum	METHOD
pressed	METHOD
radiator	CLASS
allowed temperature	MWT_GLOBAL_METHOD
first sensor	MWT_FIELD
block	METHOD
continue	GLOBAL_METHOD
pause	METHOD
pricked	METHOD
button	CLASS
pace	METHOD
turned on	MWT_METHOD
the power switch	MWT_CLASS
conclude	METHOD
sparked	METHOD
pressure sensor	MWT_CLASS
switch	CLASS
laser	CLASS
second sensor	MWT_FIELD
set	BOOL_COND_METHOD
boiler	CLASS
times triggered	MWT_FIELD
initiate	METHOD
triggered	METHOD
button one	MWT_FIELD
next process	MWT_GLOBAL_METHOD
temperature sensor	MWT_CLASS
rpm value	MWT_GLOBAL_METHOD
stove	CLASS
next task	MWT_GLOBAL_METHOD
pushed down	MWT_METHOD
launched	METHOD
active	METHOD
sensor	CLASS
activate	METHOD

effected	METHOD
rate	METHOD
store	GLOBAL_METHOD
down	METHOD
started	METHOD
moved	METHOD
commenced	METHOD
initiated	METHOD
temperature	GLOBAL_METHOD
triggers	METHOD
commence	METHOD
pushed	METHOD
sensor one	MWT_FIELD
stop	METHOD
function two	MWT_GLOBAL_METHOD
first button	MWT_FIELD
the power button	MWT_CLASS
sound sensor	MWT_CLASS
deactivate	GLOBAL_METHOD
start	METHOD
magnetic sensor	MWT_CLASS
rpm	GLOBAL_METHOD
proximity sensor	MWT_CLASS
is on	MWT_RETURN_METHOD
position sensor	MWT_CLASS
second button	MWT_FIELD
show	METHOD
actuator	CLASS
camera	CLASS
photodiode	CLASS
tripped	METHOD
affected	METHOD
light sensor	MWT_CLASS
second function	MWT_GLOBAL_METHOD
function one	MWT_GLOBAL_METHOD
launch	METHOD
temperature threshold	MWT_GLOBAL_METHOD

the sensor	MWT_CLASS
is off	MWT_RETURN_METHOD
velocity	METHOD
activate	GLOBAL_METHOD
kill	METHOD
sensor two	MWT_FIELD
gyroscope sensor	MWT_CLASS
acceleration sensor	MWT_CLASS
detector	CLASS
close	METHOD
clicked	METHOD
deactivate	METHOD
button two	MWT_FIELD
speed	METHOD
the switch	MWT_CLASS
inactive	RETURN_METHOD
motor threshold	MWT_GLOBAL_METHOD
humidity sensor	MWT_CLASS
turned	RETURN_METHOD
turn	BOOL_COND_METHOD
save	GLOBAL_METHOD
this	CLASS
the detector	MWT_CLASS
power switch	MWT_CLASS
motor	CLASS
cease	METHOD
heater	CLASS
display	METHOD
first function	MWT_GLOBAL_METHOD
activated	METHOD