Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

Artificial Intelligence and Computer Science

Voice-Driven Web-Based Code Editor

BACHELOR THESIS

Author: Cyril Janeček

Supervisor: Doc. Ing. Petr Pollák, CSc.

Date: May, 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Janeček Cyril**	Personal ID number:	**491992**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Voice-Driven Web-Based Code Editor

Bachelor’s thesis title in Czech:

Hlasem ovládaný webový editor kódu

Guidelines:

1. Learn the basics of Automatic Speech Recognition (ASR) and compare available ASR engines usable for voice control of computer applications. Focus on availability, price, accuracy, and latency of particular engines.
2. Study the possibilities of creating an interactive web applications with the focus on an analysis of available libraries and frameworks.
3. Design and implement a prototype of WEB-based editor of source code in the language JavaScript. Take into account a possible extension of created application to other programming languages as well.

Bibliography / sources:

[1] J. E. Hopcroft, R. Motwani, J. D. Ullman: Introduction to Automata Theory,Languages, and Computation, 3rd Edition, Addison-Wesley, 2006.
[2] Lawrence R. Rabiner and Ronald W. Schafer: Introduction to Digital Speech Processing. now publishers inc., 2007.
[3] Google Cloud - Speech-to-Text [online]. Dostupné z: https://cloud.google.com/speech-to-text

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Petr Pollák, CSc.   Department of Circuit Theory FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **12.12.2021**   Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

doc. Ing. Petr Pollák, CSc.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt   Student’s signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, May 20, 2022 ....

Cyril Janeček

Acknowledgment

I would like to thank my supervisor, Doc. Ing. Petr Pollak, CSc., for all the invaluable advice and guidance throughout the year.

Cyril Janeček

Title:
Voice-Driven Web-Based Code Editor

Author:	Cyril Janeček
Study programme:	Open informatics
Specialization:	Artificial Intelligence and Computer Science
Publication type:	Bachelor thesis
Supervisor:	Doc. Ing. Petr Pollák, CSc. Department of Circuit Theory
Key words (English):	web accessibility, voice control, formal language, code parsing
Key words (Czech):	přístupnost webu, hlasové ovládání, formální jazyk, parsování kódu

Abstract (English): This thesis describes the implementation of a voice-driven code editor that enables injured or disabled people to write programs using their voice. Existing working solutions are often paid or require a complicated setup, making them less accessible. This thesis aims to implement a working prototype of such an editor as a web application, with a main focus on accesibility and ease of use. A free version of Google cloud's speech-to-text service is used for speech recognition. The transcribed words are then processed by lexical and syntactical analysis, which transform them to valid JavaScript code. For this transformation, a custom context-free grammar and a parsing algorithm are implemented that also enable additional advanced features of the editor such as smart identifier resolution, undo/redo functionality, and automatic indentation. The resulting application has been tested by four users, which have provided mostly positive feedback. The code editor, as well as its source code, are available publicly on Gitlab.

Abstract (Czech): Tato práce popisuje implementaci hlasově ovládaného editoru kódu, který umožňuje lidem s pohybovým omezením programovat pomocí svého hlasu. Existující aplikace jsou často placené, nebo vyžadují poměrně komplikovanou instalaci, což snižuje jejich přístupnost. Tato práce si klade za cíl implementovat fungující prototyp takového editoru jako webovou aplikaci, jejíž hlavní zaměření je přístupnost a jednoduchost použití. Bezplatná verze služby Google cloud's speech-to-text je použita pro rozpoznávání řeči. Přepsaná slova jsou poté zpracována lexikální a syntaktickou analýzou, jež je transformují na kód jazyka JavaScript. Pro transformaci je použita vlastní implementace bezkontextové gramatiky a parsovacího algoritmu, které také umožňují další pokročilé funkce editoru jako například automatickou indentaci, rozpoznání identifikátorů a možnost vracet změny editoru. Výsledná aplikace byla otestována čtyřmi uživateli, kteří poskytli převážně pozitivní zpětnou vazbu. Editor i jeho zdrojové kódy jsou veřejně dostupné ve službě Gitlab.

Contents

1 Introduction

The topic of accessibility has been gaining a lot of attention in recent years, with associations such as the World Health Organization and the United Nations raising awareness of the issue [2, 3]. The purpose of accessibility is to enable people with disabilities to access information and to interact with services, products, and tools [4].

With recent advancements of digital technology, there has been a significant increase in the number of accessibility solutions. Specifically, improvements in speech recognition and synthesis allowed various new tools and products to emerge, such as automatic captioning, text readers, voice-controlled devices, and many more.

Having suffered an arm injury, I was looking for a way to increase my one-handed programming speed using speech recognition. Personal experiments quickly showed that general dictation applications are unviable for this purpose. Some specialized solutions exist, such as Code-by-voice and Dictation-toolbox. However, they require a third-party recognition system as well as a complex setup and configuration [5, 6].

There is a free application, called Serenade, that enables coding by voice by providing plugins for popular code editors. Its code dictation works relatively well, however, it also has a few downsides. The setup consists of several time consuming steps such as downloading and installing the app itself, installing a code editor plugin and creating an account. Furthermore, its codebase is closed-source and there is no description of the implementation [7].

This led to the idea of creating similar software as a web application. As a result, it would require no setup, allowing users to simply visit the webpage and start dictating code. Thus, the goal of this thesis is to implement a functional prototype of such a voice-controlled code editor. By making its codebase open-source and providing detailed description of the implementation within this thesis, I aim to provide a foundation for further development in this area of accessibility.

The opening chapter gives a brief overview of two approaches for designing speech recognition systems. Available solutions are also discussed. The following chapter explains the basics of formal language theory necessary for understanding core parts of the editor's implementation, which is thorougly described in the next chapter. The final chapter summarizes the results of user testing and discusses possible options for future work.

2 Automatic speech recognition

Automatic speech recognition (ASR) is the process of converting a speech signal into a text representation of the spoken words [8]. This chapter briefly describes two commonly used approaches in ASR system design. Next, a few existing solutions are discussed and compared.

2.1 HMM-based recognition systems

The traditional recognition systems are usually based on Hidden Markov Models (HMM). These systems consist of multiple logical blocks, which are shown in a simplified diagram in Figure 2.1. The feature analysis block first converts the input speech signal $s[n]$ into a sequence of feature vectors X. Next, the pattern classification block decodes X into a maximum likelihood string $\hat{W}$. It uses a set of acoustic models (represented as HMM), a word lexicon, and a language model to assign a match score to each proposed string. The final block then provides a confidence score for each of the individual words in $\hat{W}$ [8].

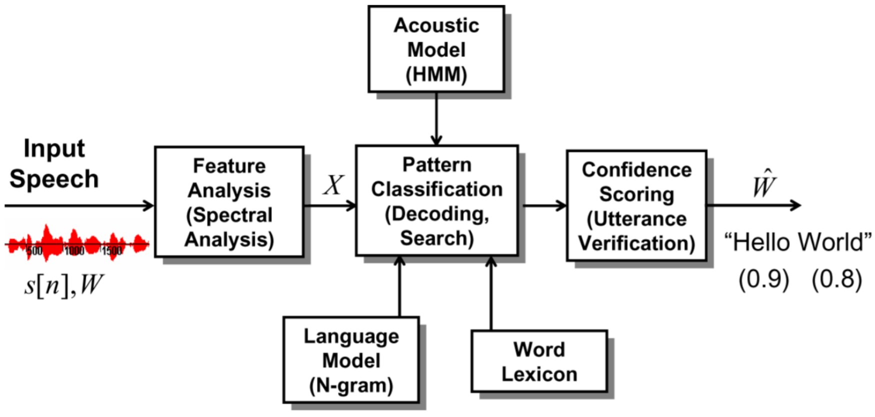

Acoustic Model (HMM)
Input Speech
$s[n], W$
Feature Analysis (Spectral Analysis)
X
Pattern Classification (Decoding, Search)
Confidence Scoring (Utterance Verification)
$\hat{W}$
"Hello World"
(0.9) (0.8)
Language Model (N-gram)
Word Lexicon

Figure 2.1: Block diagram of an overall speech recognition system [8]

2.1.1 Feature analysis

The input signal is usually split and processed one segment at a time. The feature analysis block first samples and quantizes the segment. Then, pre-emphasis is used to compensate for the decreased signal power of the higher frequencies. The pre-emphasized signal is separated into frames and converted to Mel-frequency cepstral coefficients. Most commonly, a standard DFT-based approach with an additional filter bank is used [8].

2.1.2 Pattern classification

The task of automatic speech recognition can be represented as a Bayes maximum a posteriori probability estimation (MAP) [8]:

$$\hat{W} = \underset{W}{\operatorname{argmax}}\, P(W|X) \tag{2.1}$$

The objective is to find a string $\hat{W}$ that maximizes the a posteriori probability of W given X. Using the Bayes rule, equation (2.1) can be rewritten as:

$$\hat{W} = \underset{W}{\operatorname{argmax}}\, \frac{P(X|W)P(W)}{P(X)} \tag{2.2}$$

Since $P(X)$ is independent of variable W, which is being optimized, equation (2.2) can be further rewritten as:

$$\hat{W} = \underbrace{\underset{W}{\operatorname{argmax}}}_{\text{Part 3}}\, \underbrace{P(X|W)}_{\text{Part 1}}\, \underbrace{P(W)}_{\text{Part 2}} \tag{2.3}$$

Here, part 1 is the probability of speech sounds for a given sentence W. This is computed by the acoustic model. Part 2 is the a priori probability of a given sentence and is computed by the language model. Part 3 represents the search through all valid sentences in order to find the maximum likelihood sentence $\hat{W}$ [8].

Acoustic model

The Acoustic Model in ASR systems is most commonly represented by an HMM [8]. This abstraction is used to model the probabilities of sequences of speech elements. Usually, elements known as triphones are used, representing individual phonemes in the context of their predecessors and successors. Typically, the so-called Gaussian mixture models (GMM) are used to model probabilities inside of the HMM. This approach is also called GMM-HMM. Alternatively, the GMM can be replaced by a deep neural network (DNN). A standard feedforward NN can be used as well as a more complex type such as a convolutional or recurrent NN [9].

Language model

The language model assigns probabilities to sequences of words based on their language context. There are many valid approaches for building a language model, the most common one being a statistical *n-gram* word model. This approach assumes that the probability of a given word in a sentence is dependent only on the $n - 1$ previous words. The probability is then estimated by counting the relative frequencies of n-tuples of words in the training set [8].

The simple 1-gram version is sometimes used, which specifies only a list of single words along with their probabilities. An even simpler model, 0-gram, assumes that all the words in the list have a uniform probability of appearing. However, bigrams and trigrams are most commonly used for continuous speech recognition.

2.2 End-to-end systems

As is shown in the previous section, the design and implementation of HMM-based ASR systems requires a significant amount of human effort and expertise. For this reason, an alternative approach to automatic speech recognition, called an end-to-end system, has been gaining a lot of popularity. Many publications such as [10] and [11] have shown that this approach can compete and potentially overcome the traditional HMM-based approach in terms of both speed and accuracy.

An end-to-end approach aims to replace all the logical blocks of the traditional system with a single neural network. The network usually takes an audio signal directly as input and outputs its corresponding text representation. Since the individual logical blocks and comprehensive pre-processing are not needed in end-to-end systems, they are significantly easier to implement and maintain than the traditional HMM-based systems. However, training neural networks requires a large amount of data, which is not always readily available [9].

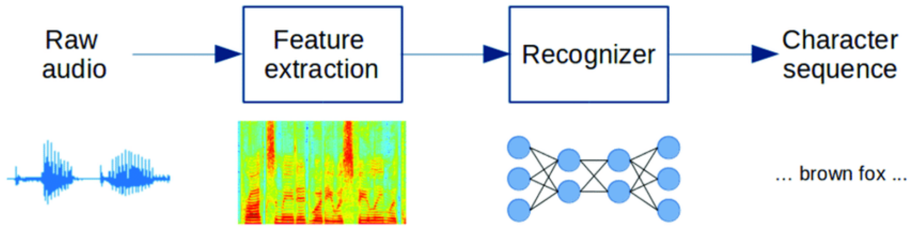


Figure 2.2: A general architecture for an end-to-end speech recognition system [12]

2.3 Available solutions

There are many available toolkits that make the implementation of ASR systems more manageable. They allow users to design, implement, and train their own speech recognition models with relative ease. Some examples of such open-source projects include Kaldi and CMUSphinx [13, 14].

One of these toolkits could be used to create a specialized ASR system designed specifically to recognize spoken programming code. Such a system could potentially improve the recognition accuracy of terms specific to programming languages. However, it would require a collection of spoken and written programs as training data, which is not currently available. However, this kind of data could potentially be collected in the future within the application if users consent to it.

Instead, a pre-trained model distributed as an online service is used in the current implementation. Again, many such solutions exist. However, this thesis focuses its research on Google cloud's speech-to-text and the Web Speech API.

2.3.1 Google cloud speech-to-text

The speech-to-text service provided within the Google cloud platform is one of the most widely used ASR services available. It uses advanced neural networks trained on vast amounts of data that Google collects through its various products. Thanks to this, the service provides high-accuracy transcription and a huge variety of features, such as automatic noise reduction, model customization, speaker diarization, punctuation, and many more.

The service provides 60 minutes of free transcription per month. The price of additional transcription depends on whether the user agrees to Google collecting their data. The cheapest option is charged at $0.016 per minute. Additionally, every transcription request is rounded up to 15 seconds, which makes it less suitable for short bits of recognition, such as voice commands [15].

2.3.2 Web Speech API

The Web Speech API is a W3C [1] supported specification that allows users to access speech recognition directly from the browser. The speech recognition in this API is supplied by browser vendors using online services or the ASR engine built into the operating system. However, only Google Chrome on desktop and Android currently provides full support for this API. When used in Chrome, the API is in fact a simplified version of the Google cloud speech-to-text. Its use is free without limitations, however, it contains fewer features than the paid version [17].

The features provided by the API include:

- Low-latency transcription with interim results
- Recognition results with alternatives ranked by their probabilities
- Specifying custom words to be recognized with increased priority

2.3.3 Comparison

Google cloud's full speech-to-text provides many useful features. However, since the voice control of the editor relies primarily on transcription accuracy and low latency, the more advanced features are in fact unnecessary. The free version of Web Speech API is thus sufficient for the use case of this application. Its implementation is described in section 4.4.

[1]World Wide Web Consortium is the leading international standards organization for the World Wide Web [16].

3 Formal languages

The theory of formal languages provides the tools necessary to process the outputs of the ASR. First, a few base terms and concepts have to be defined. An *alphabet* Σ is a finite, non-empty set of symbols such as $\{a, b, c\}$. A *string* or *word* is a finite sequence of symbols chosen from an alphabet, such as *cab*. An empty string is denoted ϵ. The set of all strings over an alphabet Σ is denoted Σ^* [18].

A *language* L is a set of specific strings chosen from some Σ^*, it is said that it is a language over Σ. A couple examples of languages include English, JavaScript, and the simple language of all words ending with b. Each language has its own alphabet Σ.

A *grammar* generating a language L is a device that produces all of the strings that are part of L and nothing else. The Chomsky hierarchy divides grammars and the languages they generate into four types [19]:

- Type 0: Recursively enumerable
- Type 1: Context-sensitive
- Type 2: Context-free
- Type 3: Regular

Each type is a superset of the following, higher number type. This thesis concerns itself only with context-free grammars, which are described in more detail in the following section.

3.1 Context-free grammars

A grammar is defined as an ordered 4-tuple $G = (N, \Sigma, S, P)$. N is a finite set of *nonterminals* and Σ is an alphabet, a finite set of *terminals*. Terminals make up the words the grammar can generate, while nonterminals can be expanded using production rules into sequences of terminals and nonterminals. One of the nonterminals is defined to be a *start symbol* S.

P is a finite set of *production rules* $\alpha \to \beta$, where α, β are strings over $N \cup \Sigma$ and α contains at least one nonterminal. The production rules of context-free grammars (CFGs) are further restricted to be $A \to \beta$, where $A \in N$. That is, the left side of the production rule is only a single nonterminal.

Listing 3.1: An example CFG generating arithmetic expressions [20]

```
G = (N, Σ, S, P)
N = {S, T}
Σ = {+, −, ∗, /, num}
P:
   S → T | T + S | T − S
   T → num | num ∗ T | num / T
```

The process of *deriving* strings by applying production rules is denoted by a right arrow $\Rightarrow$. Suppose $G = (N, \Sigma, S, P)$ is a CFG. Let $\alpha A \beta$ be a string, where $A \in N$ and $\alpha, \beta \in (N \cup \Sigma)$. Let also $A \to \gamma$ be a production rule of G. Then $\alpha A \beta \underset{\mathrm{G}}{\Rightarrow} \alpha\gamma\beta$ is a valid derivation. If the grammar being referred to is known, the G symbol under the arrow can be omitted. The notation $\alpha \Rightarrow^{*} \beta$ represents zero or more derivation steps. A *leftmost derivation* replaces at each step the leftmost nonterminal, whereas a *rightmost derivation* replaces the rightmost nonterminal.

Listing 3.2: An example leftmost derivation using the CFG from 3.1

```
S ⇒ T + S ⇒ num + S ⇒ num + T
  ⇒ num + num / T ⇒ num + num / num
```

The language of a grammar $G = (N, \Sigma, S, P)$ is denoted as

$$L(G) = \{w \in \Sigma^{*} | S \underset{\mathrm{G}}{\Rightarrow}^{*} w\} \tag{3.1}$$

It is said that the grammar G generates the language L. If G is a CFG, then L is a context-free language (or CFL) [18].

3.2 Parsing

A so-called *parse tree* represents how a given string has been derived. The leaf nodes of the tree are labeled by terminals. When read from left to right, they give the derived string. The internal nodes are labeled by nonterminals. For each internal node, there must be a production such that its left side is the label of the node and the labels of its children (from left to right) form the right side of the production. A grammar G is said to be *ambiguous* if, for any given string ω, there exist two different parse trees for how ω was derived by production rules of G [18].

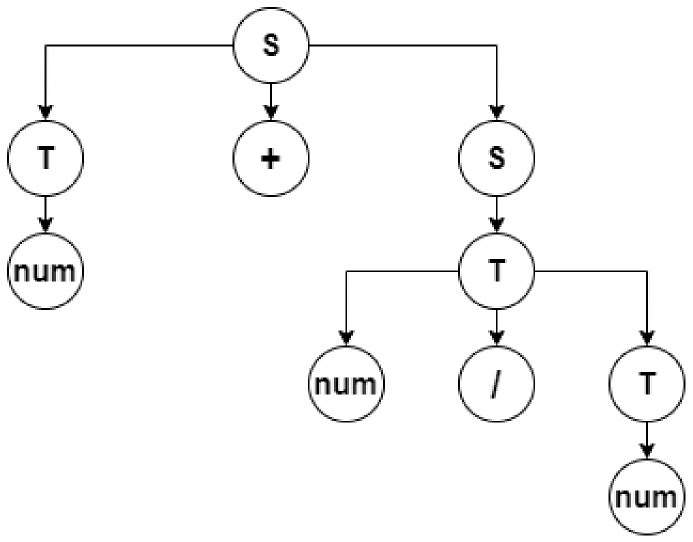


Figure 3.1: A parse tree corresponding to the derivation in 3.2

Parsing is the process of recovering a parse tree for a given string. There are many parsing methods; however, this thesis focuses on LL(k) parsing. This method of parsing reads the input from left to right and, at each step, uses the leftmost derivation. The *k* means how many tokens ahead the parser can look when deciding which production rule to use in a derivation. This thesis uses an LL(1) parser, which requires its grammar to be in LL(1) form. There are necessary but not sufficient conditions for LL(1) grammars. They have to be unambiguous, and they must have no *left-recursive* production rules. That is, rules of the type $A \rightarrow A\gamma$, where $A \in N$ and $\gamma \in (N \cup \Sigma^*)$ for a grammar $G = (N, \Sigma, S, P)$ [20].

4 Implementation

This chapter describes the implementation of a usable first version of the application. The main focus is on source code quality, architecture, and overall application design. This should allow for further iterative expansion and improvement of the version finished within the scope of this thesis.

4.1 Organization

A software called git [21] is used to version the source code. New features can thus be developed in isolated branches and are only merged into the main branch when their code is appropriately tested. Within the branches, code changes are bundled in so-called commits that can also be reverted when necessary. Additionally, it is possible to switch the source code to any previous version. This helps during debugging to narrow the cause of a bug down to a single commit.

The versioned source code is hosted at `https://gitlab.com/crispjam/speechcode` by a service called Gitlab [22]. This open-source software provides a place to store the code as well as other valuable features such as branch creation, merging of branches, issue tracking, and many more. Gitlab also provides a platform for publicly hosting static websites, which is used for the production version of the code editor, hosted at `https://crispjam.gitlab.io/speechcode/`. The deployment is configured to be triggered by every commit to the main branch, keeping the hosted application up-to-date with the source code.

Another web service that helps with implementation organization is called Trello [23]. It is used to track the development life cycle of new features, bugs, and code refactorings. It provides a visual overview of the current priorities and incentivizes one to break up large features into smaller tasks, making the development process more organized.

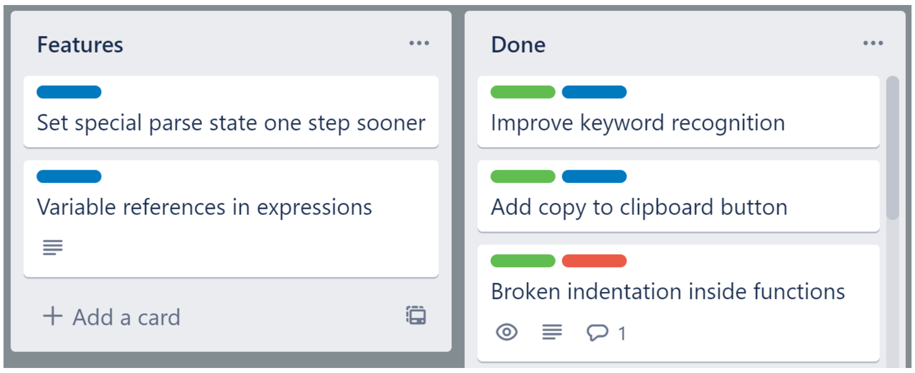

Features
Set special parse state one step sooner
Variable references in expressions
+ Add a card
Done
Improve keyword recognition
Add copy to clipboard button
Broken indentation inside functions
1

Figure 4.1: Two of the task lists inside the project's Trello board

4.2 Architecture

The code editor is designed to be a React-based front-end only application. React [24] is a JavaScript library for building User Interfaces (UIs). It introduces a special syntax called JSX, which blends HTML and JavaScript together, making the UI's visuals and logic more inter-connected. It also allows the programmer to create reusable and self-contained components avoiding copying and pasting any of the application's code. The most significant advantage of React, however, is its efficiency. It tracks the application state and re-renders only the parts of the UI that have currently changed.

Additionally, Typescript [25] is used, which is a strict syntactical superset of JavaScript. Before execution, it is transpiled into JavaScript, which means that it runs anywhere JavaScript would. It provides a syntax for strong typing, helping to reveal errors early in the development process. In my experience, it also makes the source code more structured and understandable because the types convey helpful information that would otherwise have to be explained in comments.

A few more open-source packages are used, most notably, Lexer, CodeMirror, and Material UI [26, 27, 28]. These are all described in more detail later in the thesis. All the packages are managed by the Node package manager npm [29].

During development, a tool called Create React App [30] is used to facilitate easy testing and debugging. Whenever the source code changes, it efficiently creates a new build, deploys it to a local server and opens a browser to the given port. Thanks to this, new changes can be tested quickly without manually rebuilding the source code.

4.3 Source structure

The application's source code is divided into two main parts. The UI React components define the content to be rendered and handle the user's input. The so-called library classes provide the underlying functionality necessary for the application. These two parts are connected by the `App` class. This React component defines the layout of its UI sub-components, handles events fired from them, and uses the library classes to produce output for the user.

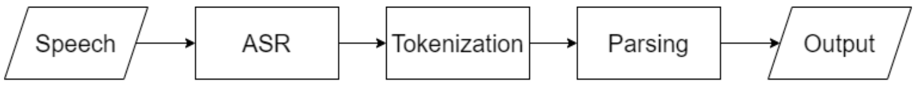


Figure 4.2: A block schema of the library part of the application

4.4 Speech recognition

The Web Speech API is used for speech recognition and is handled by two classes. The `WebspeechApiWrapper` class configures the API and abstracts its specific functionality. Thanks to this, upon replacing the ASR core with a different provider, only this wrapper class would have to be modified, and the rest of the application could remain unchanged. That is, if the new ASR provides the necessary features used in the application. The `Recognizer` class receives results from `WebspeechApiWrapper` and passes them onto the `App` class. It also handles starting and stopping of the recognition.

The Web Speech API is configured by providing a list of words whose importance in the recognition should be emphasized. The list consists of the expected keywords used within all parts of the application. The configuration of these keywords seems to have improved their recognition accuracy. However, no rigorous tests have been made to confirm this.

During recognition, as long as the user is speaking, the API continuously provides interim results that are displayed in real-time by the `RecognitionFeedback` component. User testing has shown that displaying these interim results makes the application feel more responsive and easy to use.

Once the user pauses, the API outputs a final transcript of the entire utterance. Due to the broader context available, the final result is usually much more accurate than the shorter interim ones. Furthermore, the final result consists of up to 10 alternative transcripts sorted by their probabilities. Testing showed that the results usually consist of between 1 - 3 alternatives. These alternatives are returned to the `App` class, which performs lexical analysis on them.

4.5 Lexical analysis

Lexical analysis (or tokenization) transforms the input text into lexical tokens. An open-source package called Lexer [26] is used for this task. Its functionality is abstracted by the `JsLexer` class, which also configures Lexer for the specifics of spoken JavaScript. A list of regex rules is defined. The first two rules remove whitespace, punctuation, and unwanted special characters from the input text. While the rest of the rules each match a regex pattern to a given token. The order of the rules matters, as it specifies their precedence.

Listing 4.1: The first three regex rules configured for Lexer

```
addRule(/\s/, () => {});
addRule(/['"‘,:!@#$%^&()~;{}[\]]/, () => {});

addRule(/[0-9]+(\.[0-9]+)?/, (lexeme) => {
  return(new JsToken('tNumber', lexeme));
});
```

Each token consists of a type, an identifying string starting with lowercase t (e.g. *tWord*) and its text content (e.g. *counter*). Upon calling the `scan` method, the Lexer package tokenizes the given input according to the specified rules and returns the resulting sequence of tokens to the `App` class.

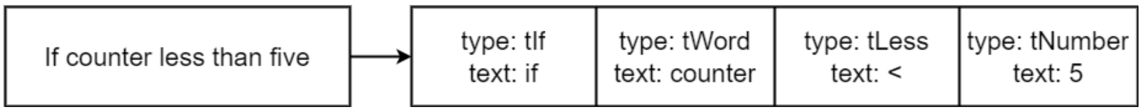


Figure 4.3: An example of tokenization

4.6 Syntactical analysis

Many existing open-source solutions for syntactical analysis are available, such as ANTLR, Jison, and PEG [31, 32, 33]. However, special behavior is needed during the parsing process in the particular use case of converting spoken JavaScript into regular JavaScript. For this reason, an entirely custom implementation is necessary.

A number of algorithms exist for parsing languages with different complexity and various degrees of expressive power. An LL(1) top-down table-driven parser has been chosen for its relative simplicity, which makes debugging of the overall application more manageable. Its functionality is described in more detail in the following subsections.

4.6.1 Grammar

The formal language of spoken JavaScript tokens is fully described by a context-free grammar. A general `Grammar` class is implemented to provide the necessary functionality that is shared for all possible CF grammars needed. The grammar specific to JavaScript is implemented in the `JsGrammar` class, whose most important part are its production rules. Each rule consists of a nonterminal on the left-hand side and one or more productions on the right-hand side. Productions consist of terminals, nonterminals, and special *plaintext* symbols. The terminals are identical to the tokens produced by Lexer in the previous step. The plaintext symbols have no meaning in formal languages and were implemented specifically for this use case. They are used during the parsing process itself to enrich its output with additional text.

Listing 4.2: A sample of JavaScript grammar rules

```
nStatements → nStatement nStatements
              | tEpsilon
nStatement  → tIf nExpr nBlock
```

The rules in the Grammar class are described by an object of key, value pairs. Each key is a nonterminal symbol. Each value is a list of productions, where a production is a list of symbols. A symbol is a string starting with an identifying lowercase letter: 'n' for nonterminals, 't' for terminals, and 'p' for plaintext.

Listing 4.3: JsGrammar rules corresponding to those defined in listing 4.2

```
nStatements: [[ 'nStatement', 'p\n', 'nStatements'],
               [ 'tEpsilon' ],
              ],
nStatement: [[ 'tIf', 'p (', 'nExpr', 'p)', 'nBlock' ]]
```

4.6.2 Parse table

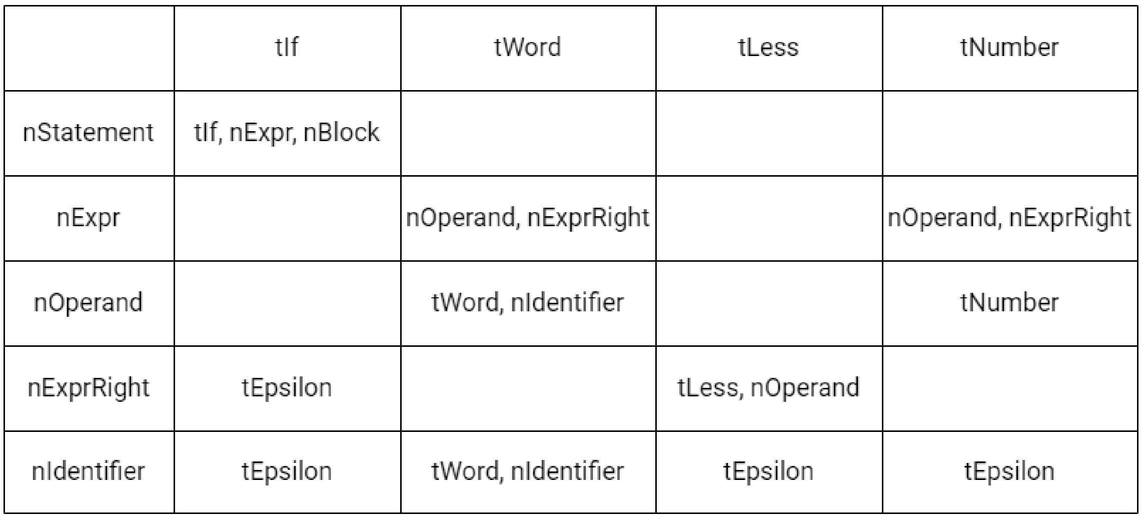
Before parsing can occur, it is necessary to construct a *parse table*. The left header of the table is made up of terminals and the upper header of nonterminals. Each cell is a right side of a production corresponding to the given nonterminal, terminal pair. The table shows which production to predict when expanding a given nonterminal and a given terminal is the first symbol of input. To efficiently construct a parse table, the first and follow sets of nonterminals need to be computed.

For a given nonterminal A, its first and follow sets are given by

$$First(A) = \{t|A \Rightarrow^* t\omega\} \tag{4.1}$$

$$Follow(A) = \{t|S \Rightarrow^* \alpha A t\omega\} \tag{4.2}$$

Here, t is a terminal, S is the start symbol of the grammar, and α, ω are arbitrary strings. Informally, $First(A)$ is the set of terminals that can appear at the beginning of string ω produced by nonterminal A. $Follow(A)$ is the set of terminals that can appear after A in any derivation [20].



	tIf	tWord	tLess	tNumber
nStatement	tIf, nExpr, nBlock			
nExpr		nOperand, nExprRight		nOperand, nExprRight
nOperand		tWord, nIdentifier		tNumber
nExprRight	tEpsilon		tLess, nOperand	
nIdentifier	tEpsilon	tWord, nIdentifier	tEpsilon	tEpsilon

Figure 4.4: A simplified example of a parse table

To construct a parse table T, one first goes through all nonterminals N and all terminals t of the gramamar and sets $T[N][t]$ to be empty. Then, for each production rule $N \to \omega$:

1. For each terminal $a \in First(\omega)$, add ω to $T[N][t]$
2. If $\epsilon \in First(\omega)$ then for each terminal $b \in Follow(N)$, add ω to $T[N, b]$

The parse table is implemented as an object of key, value pairs in the source code. The key is a given nonterminal N. The value is another object, whose key is a terminal t and its value is a list of symbols of production ω.

4.6.3 Parsing

The two main components of the parser are a stack and the parse table. The stack is initialized in the constructor of the `Parser` class to contain only the start symbol of the given grammar. The parse table is built according to the steps explained above. Note that during parsing, the parse tree for the given input is recovered implicitly but is not saved as it is not necessary for the purposes of the application.

After the `App` class receives a list of tokens from `JsLexer`, it calls the `parseInput` method of `Parser`. This method first initializes `textOuput` to be an empty string and then passes through the input tokens (terminals). At each step, the parser looks at the first terminal of input `t` and the top of the stack `X`. It then consults the parse table `T` in the following way:

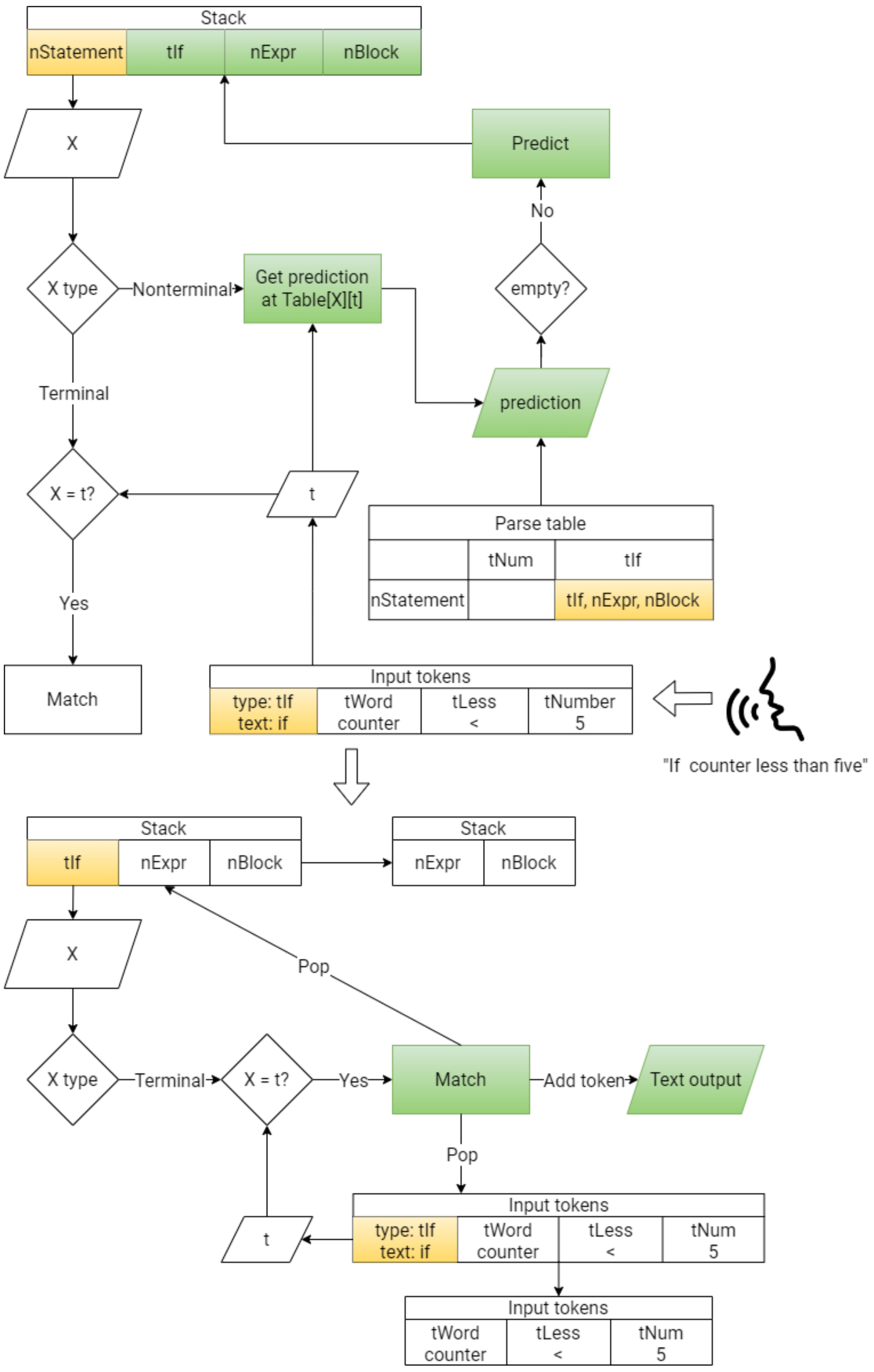


Figure 4.5: A diagram depicting two concrete steps of parsing the phrase 'If counter less than five'

1. If X is a nonterminal, then prediction = T[X][t], if prediction is empty, the input is syntactically invalid, and a parse error is raised. Otherwise, add prediction to the top of the stack.
2. If X is a terminal and t == X, then match, adding the text of X to the textOutput and pop the top of the stack and input.
3. If X is a plaintext symbol, then add its text to the textOutput, the stack stays unchanged.

If the method goes through all of the input without an error, the current version of the stack is saved, and textOutput is returned back to the App class.

Thanks to the plaintext symbols, the grammar rules can have text in the productions that only appears in the output and is not expected in the input. Thanks to this, the user does not have to dictate redundant symbols such as curly braces around code blocks.

4.7 Semantic analysis

In compilers, the next step after syntactical analysis is usually semantic analysis. Its primary purpose is to ensure that all types in the program are valid and that identifiers are properly declared before reference [20]. For this application, a simplified version of semantic checking is used to enhance the user experience.

The Parser class stores an instance of SemanticContext. This class keeps track of the currently declared identifiers and their scope (or block-level). Identifiers are stored in a list of key, value objects, where the key is the name of the identifier, and the value is a number identifying its block level. Whenever the Parser encounters a nonterminal called nNewId on its stack, it enters a new state called newId. While in this state, the parser collects tokens corresponding to the identifier in its parseInput method. Upon encountering a token that no longer corresponds to the identifier, it exits the special state, adding the complete identifier to textOutput and saving it to SemanticContext along with the current block level.

A similar process takes place for identifier reference. When the Parser encounters a nonterminal called nExistingId on its stack, it enters a new state called existingId. The parser again collects tokens corresponding to this identifier until encountering a token that no longer belongs there, at which point it exits the special state. Then, SemanticContext returns its existing identifiers in a list sorted by similarity to the text of the referencing identifier. The `Parser` class then chooses the most similar one as the actually referenced identifier.

When `Parser` encounters a terminal called `tBlock`, `SemanticContext` increases the block level and indentation of the resulting text. Similarly, encountering `tFinishBlock` decreases block level and indentation. `SemanticContext` then goes through all the existing identifiers and removes those whose block level is larger than the new value, keeping only the identifiers for the particular scope.

This ensures that the user does not reference undeclared identifiers. It also helps to reference the closest identifier available when the speech recognition makes a slight error.

4.8 Displaying results

The `App` class goes through the speech recognition variants it receives from `Recognizer`. It performs lexical, syntactical, and semantic analysis, and if it encounters a parse error, it moves on to the following variant. If the parse finishes without errors for any of the variants, the resulting text is inserted into a code editor, and the given variant is marked as accepted, throwing the remaining ones away. The text of the accepted variant is then shown in green by the `RecognitionFeedback` component. On the other hand, if none of the variants is parsed without an error, nothing is inserted into the code editor, and the text of the first variant is displayed in red by `RecognitionFeedback`. The interim results given by `Recognizer` are shown in the same place in gray.

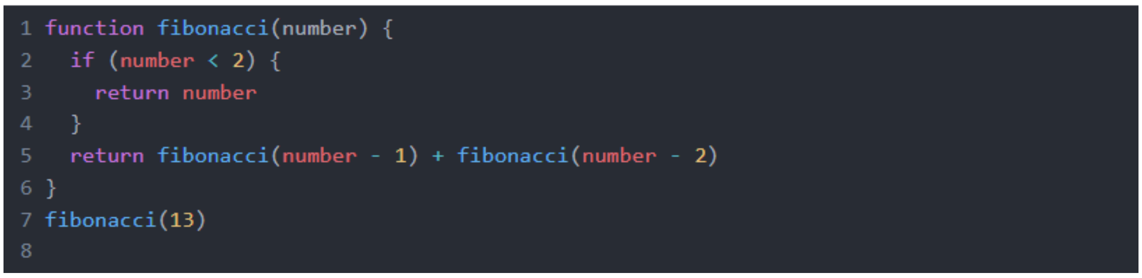
4.8.1 Code editor

For the first prototypes of the application, only a simple HTML text area was used to display the resulting code. However, it soon proved to be insufficient due to the lack of functionality. There are many open-source libraries that implement online code editors. This thesis focuses on three of the most popular ones: Ace, Monaco, and CodeMirror [34, 35, 27].

Ace was one of the first popular libraries of its kind to be created. Currently, it has a wide variety of features and supports mobile devices. However, its BSD 3 license is not the most permissive. Furthermore, its development has since been deprioritized, and therefore bug fixes and new features are released infrequently.

Monaco is an editor that powers the widely popular desktop application Visual Studio Code [36]. As such, it provides even more features than Ace and has a large community of followers and contributors. It is also licensed under the very permissive MIT license. However, the number of features causes the package size to be exceedingly large. Therefore, using it in the application would increase its total size and reduce its performance. Monaco also does not support mobile devices at all. And while mobile is not the target platform for the application, it is always advantageous not to exclude any users if possible.

CodeMirror is a relatively new project that has been gaining a lot of popularity. It fully supports mobile devices, is licensed under MIT, and provides the features necessary for this application. Its design philosophy is to be as modular as possible, which means that the programmer can choose which features to use, limiting the package size to the bare minimum.



```
function fibonacci(number) {
  if (number < 2) {
    return number
  }
  return fibonacci(number - 1) + fibonacci(number - 2)
}
fibonacci(13)

```

Figure 4.6: A screenshot of the editor

For the given use case, CodeMirror is the best solution. In the application, it is configured to highlight code syntax, show line numbers, and keep track of the editor history, providing the options to undo and redo changes. Handling manual changes to the editor would require the `Parser` class to be able to parse written JavaScript and transform the resulting stack to the format of spoken JavaScript. This is not implemented in the current version, and thus manual editing of the editor is disabled.

4.8.2 Guiding the user

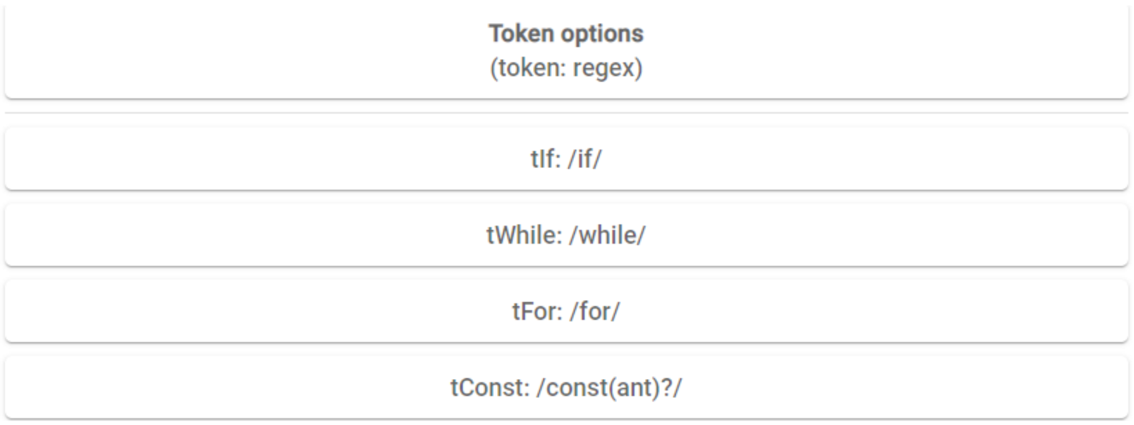
As is described above, code is inserted into the code editor only if the dictated input is syntactically valid. Because of this, the user must follow the specific syntax described by `JsGrammar`. Since this syntax differs from that of actual JavaScript, even experienced programmers need guidance on what they should dictate.

At any point during the verbal programming, what the user can say is defined by all the possible lexical tokens (or terminals) that are expected and syntactically valid in the given context. The `Parser` class, therefore, implements a method that retrieves all of these possible terminals.

The method goes through the parsing stack from top to bottom. If the current stack symbol is not a nonterminal, it advances to the next one. If the current stack symbol is a nonterminal, the method inspects the parse table at the row of this nonterminal and does the following:

1. For each column (terminal) in the row: if the cell contains a production and is therefore not empty, add the corresponding terminal to the list.
2. If any of the cells in the row contain ϵ, the nonterminal is nullable and the stack advances to the next symbol. Otherwise, end cycle.

The list of terminals is returned to `App`, which retrieves the regular expression for each terminal from `JsLexer` and displays each terminal, regex pair with the `PossibleTokensUI` component.



Token options
(token: regex)

tIf: /if/

tWhile: /while/

tFor: /for/

tConst: /const(ant)?/

Figure 4.7: An example of possible token regexes

4.8.3 Handling errors

For the purpose of this section, an error is defined as the user saying something and the application doing something else. One type of error is the `Parser` raising a syntactical error. In this case, the user either said something that is syntactically invalid or was misunderstood by the ASR. Either way, nothing is inserted into the code editor, and the user has to try again.

A different kind of error occurs when the ASR misunderstands the user, but the recognized text is syntactically correct by chance. In this case, the wrong text is inserted into the code editor, and thus the user requires a way to undo it. The CodeMirror editor provides undo/redo functionality out of the box. However, the changes must also be made in `Parser`.

For this purpose, a class called `StateHistory` is implemented. It operates on a generic type called `StateType`. The class keeps track of an undo stack and a redo stack, setting both to be initially empty. In its constructor, it receives the initial state and sets it to be the current state.

The `add` method sets the current state to the newly received state, adds it to the undo stack, and resets the redo stack to be empty. The `undo` method first checks the undo stack. If it is empty, the current state is returned without any change. Otherwise, the current state is pushed to the redo stack, and a new state is popped from the undo stack and is saved as the current state. The `redo` method flips the names of the stack but otherwise works identically.

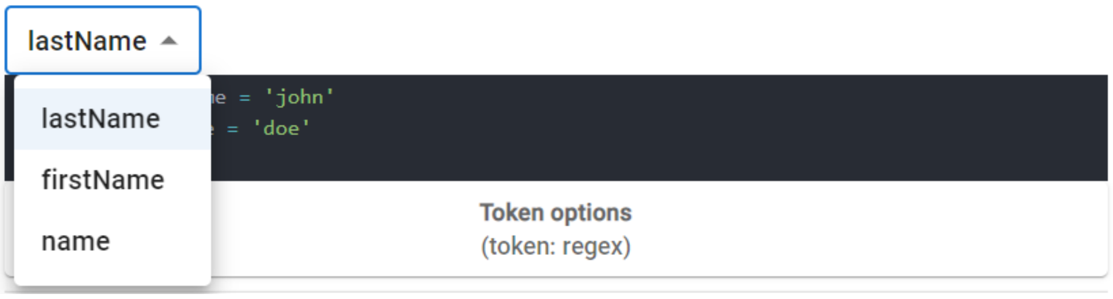
Whenever the `Parser` class successfully finishes a parse that results in a non-empty `textOutput`, it adds its stack and other state variables to its instance of `StateHistory`. The user then has the option to say either 'undo' or 'redo' or click the corresponding button, and the appropriate action is triggered for the `Parser StateHistory` as well as the editor.

4.8.4 Solving identifier problems

When the user wants to create an `if statement`, there is only one way of doing that, by saying 'if'. The ASR might misunderstand the user and instead output 'is'. In that case, the user has to try again. However, chances are that at least one of the speech recognition alternatives is correct, and since there is only a single valid option, it is accepted and all the other ones are dismissed.

However, when the user wants to create a new identifier, there is an intractable number of valid options as an identifier can consist of any combination of English lowercase characters. Because of this, it is likely that the first speech recognition alternative is immediately accepted, even though it might not be correct. Additionally, the user might want to name a variable with a word that the ASR does not know, making it impossible to do so by using speech. To provide a partial solution to this, a component called `IdentifierUI` is implemented. Whenever `Parser` enters `newId` state, a text field appears, allowing the user to manually modify the recognized identifier. When `Parser` exits the state, the value of the text field is saved as the identifier in `SemanticContext`.

Referencing identifiers is partially solved by choosing the most similar one automatically by `SemanticContext`. However, to further improve this, a selecting tool opens up when `Parser` enters `existingId` state. The user can then manually select any of the available identifiers.



```
e = 'john'
 = 'doe'
```

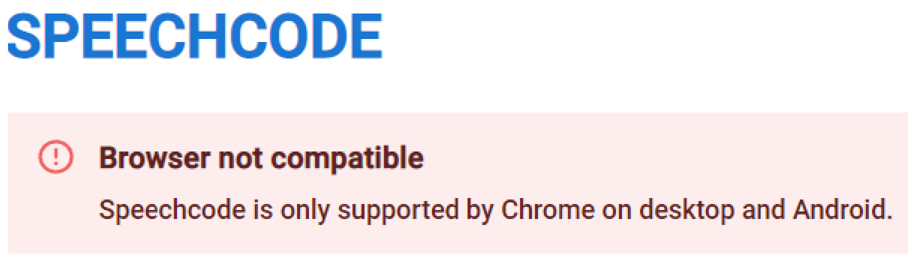
Figure 4.8: A select for choosing from existing identifiers

4.9 Limitations

As the time for implementation is not unlimited, priorities have had to be established to finish the application. The main focus has thus been on finishing a usable first version of the application, whose features work well together, rather than spending substantial time with each feature to reach its perfection. For this reason, there are a few known limitations and intentional simplifications in the current version of the application.

First and foremost, the grammar defined in `JsGrammar` does not cover the whole JavaScript language, which would be exceedingly difficult to describe by an LL(1) grammar. The implementation, therefore, intentionally omits some of JavaScript's more complex features, such as classes, regular expressions, objects, asynchronous functions, and a few more. This allows the grammar to be relatively well structured and easy to follow, making debugging much more manageable.

There are also features that have been planned out but not implemented, one of the largest ones being the connection between verbal and manual programming. That is, allowing users to switch between dictating their code and writing and modifying it manually inside the editor. This would require an additional grammar to be defined and a way for the `Parser` to be able to parse both spoken and written JavaScript and transform symbols from one grammar to the other one.


SPEECHCODE

Browser not compatible

Speechcode is only supported by Chrome on desktop and Android.

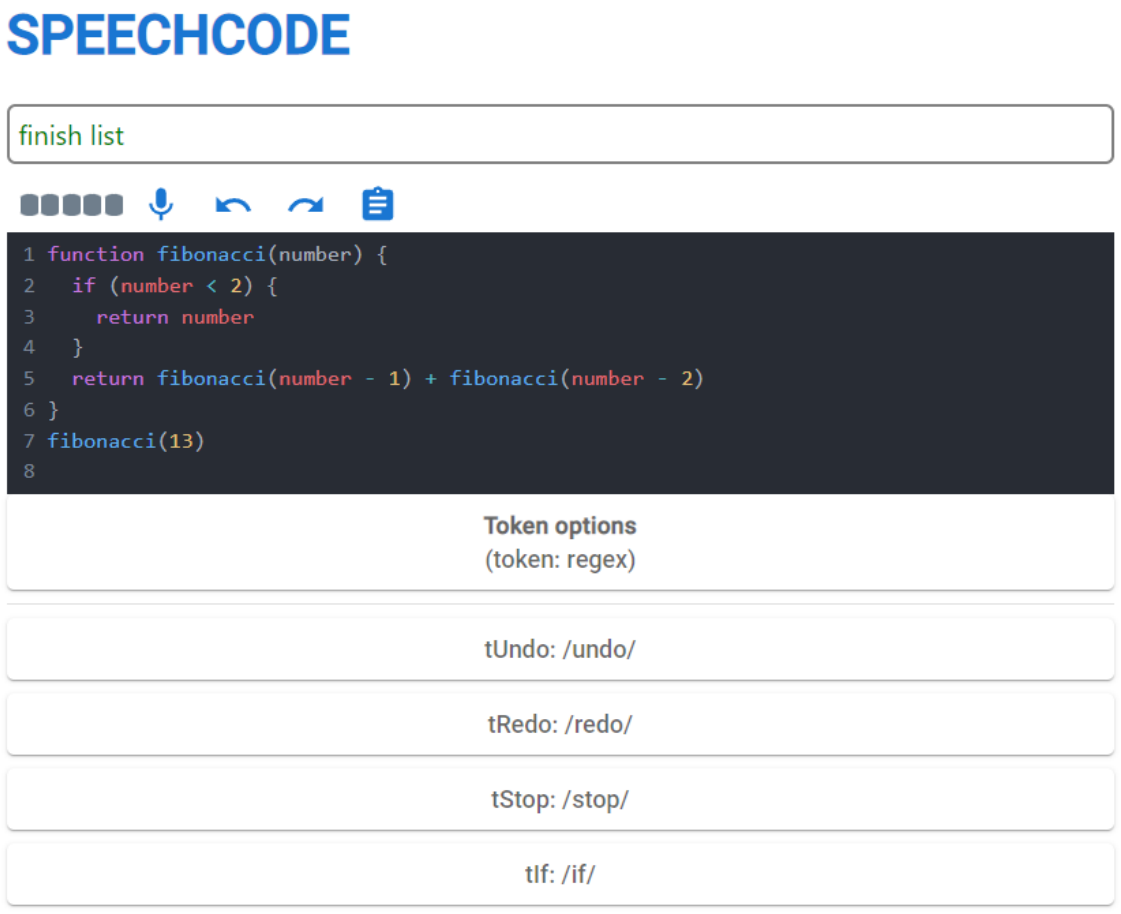
Figure 4.9: A screenshot of the application in an incompatible browser

Some existing features could also be improved. For example, the identifier selector and modifier text field work only when the user pauses their dictation at the point of naming the identifier. If they continue with the rest of the code, the identifier is composed automatically of the appropriate words, but the user does not get a chance to modify it manually. This would be partially solved by allowing users to alter the contents of the editor manually, as described above. In such a case, there would be no need for the identifier modifier text field. The select for choosing from existing identifiers could theoretically be embedded in the editor. However, that would require writing an extension package for CodeMirror.

Additionally, due to the limited support for the Webspeech API, the application runs only in Google Chrome on desktop and Android. A different ASR system would have to be used in order to support other browsers.

5 Application evaluation

The application has been brought to a point where it is usable and somewhat self-explanatory. It is hosted publicly, which makes it possible to share it with anyone with internet access. As such, it is suitable for user testing.



```
function fibonacci(number) {
  if (number < 2) {
    return number
  }
  return fibonacci(number - 1) + fibonacci(number - 2)
}
fibonacci(13)
```

Figure 5.1: A screenshot of the whole application

5.1 User testing

Four fellow programmers have been asked to test the application. Their feedback proved to be indispensable as a large number of bugs were revealed that would otherwise most likely remain undetected. There have also been a couple of main takeaways about the app's overall usability.

One of the main issues for some testers was that some of the features were somewhat hidden. The undo and redo buttons were clearly visible, but it was not apparent to a user that they could also dictate the words 'undo' and 'redo'. This has since been fixed by displaying these special commands in the list of possible terminals described in section 4.8.2.

The other feature that some users had trouble noticing is the identifier select and modifier text field. These only appear when the user pauses during describing the identifier, which some users do not do, in turn never revealing that the feature exists and that it could potentially help them. This is discussed in more detail in section 4.9.

There have also been issues with the ASR. Sometimes, a user would not know the exact pronunciation of a particular keyword, causing the ASR to misunderstand them repeatedly, making the user frustrated and eventually giving up. It has also been found that the Web speech API recognition is very sensitive to the given acoustic environment, especially surrounding noise. In a noisy environment, the ASR usually does not recognize anything at all and if it does, the accuracy is significantly reduced. Additionally, when exposed to an extended sequence of noise, the ASR gets into a broken state, no longer recognizing anything until it is restarted. This has been quite common when talking in Czech between dictating in English. The Czech confuses the ASR's context to a point that it can no longer recover from. The main issue with this is that the API does not signal about such a broken state, making it impossible to automatically restart it.

However, despite these shortcomings, all of the testers have been able to dictate simple programs with relative ease. Most have commented positively on the overall UI design, especially on the user guidance described in section 4.8.2.

5.2 Possible future work

Although the application has received quite positive user feedback, it is still far from completely replacing manual programming. However, it seems to be a good starting point to improve upon with more advanced features. As such, there are many options for potential future work.

One of the most useful additions would be allowing users to edit the editor contents manually, as it would solve several issues described in section 4.9. The user feedback regarding this feature might be skewed by the fact that none of the testers were disabled or injured. However, the application is not targeted exclusively to people requiring full voice control. This partial manual control over the editor's contents would greatly benefit people with limited motor capabilities stemming from temporary injuries or chronic conditions such as the Carpal tunnel syndrome [37].

The ASR could also be improved. It would be beneficial to do more research on the solutions available and to test and compare them extensively. Or, given enough time, a custom ASR system could be developed using one of the toolkits discussed in section 2.3. A more accurate and reliable recognition system could greatly benefit the overall usability of the application.

To accommodate the needs of more programmers, it would be necessary to expand the grammar defined in `JsGrammar` to cover all of JavaScript and to add grammars for more programming languages. The source code of the application is designed so that implementing support for a new language should be relatively straightforward. A new class deriving the general `Grammar` class would be created. Its rules would follow the same structure as in `JsGrammar` but would describe the given language instead of JavaScript. A few special cases regarding indentation and identifier reference would have to be covered, but otherwise the new programming language should work without problems.

Adding a back-end side to the web application would open up the possibility of implementing a number of valuable features. Most importantly, a database could be used to save the user's work so that it is not lost upon leaving or refreshing the site. This would also allow the implementation of project creation consisting of multiple files. However, an authentication mechanism would be required so that users could access only their own work.

Last but not least, there is a lot of room for improvement in the options for voice control. Users could benefit from the ability to modify existing code by exclusively using voice commands. This would mean adding support for line navigation, selection, deletion, and more advanced features such as smart refactoring of identifiers.

Implementing all of these features and improvements would require substantial work. It would, however, make the application a real alternative to conventional programming.

6 Conclusion

A working prototype of a voice-driven code editor has been implemented as a web application. It is hosted publicly at `https://crispjam.gitlab.io/speechcode/` and is functional in Google Chrome on desktop and Android. Its code base is available at `https://gitlab.com/crispjam/speechcode`. The thesis provides a detailed description of the editor's implementation.

ASR systems are discussed in terms of possible design approaches and available solutions. Web speech API [17] has been chosen for recognition due to its unlimited free use and a sufficient number of features. The recognized text is processed by lexical and syntactical analysis, which match the spoken words to valid JavaScript code. A custom implementation of a syntactic parser has been made to enable advanced features. These include smart identifier resolution, undo/redo functionality, and automatic indentation.

An open-source library called CodeMirror [27] is used to display the output. It provides a code editor with line numbers, code highlighting, and more. The rest of the features are wrapped by a User Interface framework called React [24].

The application has been tested by four users, who have provided mostly positive feedback. All of them were able to dictate simple programs with relative ease and most commented positively on the editor's UI and overall usability.

By thoroughly describing the editor's implementation and making its codebase open-source, the thesis should serve as a foundation for further development in this area of accesibility. Thus, it should help make programming more inclusive for a wide variety of users.

Bibliography

1. *Flaticon* [online] [visited on 2022-05-10]. Available from: `https://www.flaticon.com/`.
2. *World report on disability 2011* [online] [visited on 2022-05-10]. Available from: `https://apps.who.int/iris/handle/10665/44575`.
3. *UN Convention on the Rights of Persons with Disabilities* [online] [visited on 2022-05-10]. Available from: `https://www.un.org/development/desa/disabilities/`.
4. HENRY, Shawn Lawton; ABOU-ZAHRA, Shadi; BREWER, Judy. The Role of Accessibility in a Universal Web. In: *Proceedings of the 11th Web for All Conference*. Seoul, Korea: Association for Computing Machinery, 2014. W4A '14.
5. *Code-by-voice* [online] [visited on 2022-05-10]. Available from: `https://github.com/simianhacker/code-by-voice`.
6. *Dictation-toolbox* [online] [visited on 2022-05-10]. Available from: `https://github.com/dictation-toolbox`.
7. *Serenade* [online] [visited on 2022-05-10]. Available from: `https://serenade.ai/`.
8. RABINER, Lawrence; SCHAFER, Ronald. *Introduction to Digital Speech Processing*. Lightning Source Incorporated, 2007. Foundations and Trends in Technology.
9. JIRKOVSKÝ, Adam. *Rozpoznávání řeči s dostupnými internetovými moduly*. 2021. MA thesis. České vysoké učení technické v Praze, Fakulta elektrotechnická.
10. HANNUN, Awni; CASE, Carl et al. Deep Speech: Scaling up end-to-end speech recognition. 2014.
11. GRAVES, Alex; JAITLY, Navdeep. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In: *Proceedings of the 31st International Conference on Machine Learning*. PMLR, 2014.
12. VAZHENINA, Daria; MARKOV, Konstantin. End-to-End Noisy Speech Recognition Using Fourier and Hilbert Spectrum Features. *Electronics*. 2020, roč. 9, p. 1157.
13. *Kaldi* [online] [visited on 2022-05-10]. Available from: `https://kaldi-asr.org/doc/about.html`.
14. *CMUSphinx* [online] [visited on 2022-05-10]. Available from: `https://cmusphinx.github.io/wiki/about/`.
15. *Google Cloud - Speech-to-Text* [online] [visited on 2022-05-10]. Available from: `https://cloud.google.com/speech-to-text`.
16. *W3c* [online] [visited on 2022-05-10]. Available from: `https://www.w3.org/`.
17. *Using the Web Speech API* [online] [visited on 2022-05-10]. Available from: `https://mzl.la/3KKFluk`.
18. HOPCROFT, John; MOTWANI, Rajeev; ULLMAN, Jeffrey. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

19. CHOMSKY, Noam. On certain formal properties of grammars. *Information and Control*. 1959, roč. 2, č. 2, pp. 137–167.
20. SCHWARZ, Keith. *Compilers* [online] [visited on 2022-05-10]. Available from: `https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/`.
21. *Git* [online] [visited on 2022-05-10]. Available from: `https://git-scm.com/`.
22. *Gitlab* [online] [visited on 2022-05-10]. Available from: `https://gitlab.com/`.
23. *Trello* [online] [visited on 2022-05-10]. Available from: `https://trello.com/en`.
24. *React* [online] [visited on 2022-05-10]. Available from: `https://reactjs.org/`.
25. *TypeScript* [online] [visited on 2022-05-10]. Available from: `https://typescriptlang.org/`.
26. *Lexer* [online] [visited on 2022-05-10]. Available from: `https://github.com/aaditmshah/lexer`.
27. *CodeMirror* [online] [visited on 2022-05-10]. Available from: `https://codemirror.net/6/`.
28. *Material UI* [online] [visited on 2022-05-10]. Available from: `https://mui.com/`.
29. *Npm* [online] [visited on 2022-05-10]. Available from: `https://www.npmjs.com/`.
30. *Create React App* [online] [visited on 2022-05-10]. Available from: `https://create-react-app.dev/`.
31. *ANTLR* [online] [visited on 2022-05-10]. Available from: `https://antlr.org/`.
32. *Jison* [online] [visited on 2022-05-10]. Available from: `https://github.com/zaach/jison`.
33. *PEG* [online] [visited on 2022-05-10]. Available from: `https://pegjs.org/`.
34. *Ace* [online] [visited on 2022-05-10]. Available from: `https://ace.c9.io/`.
35. *Monaco* [online] [visited on 2022-05-10]. Available from: `https://microsoft.github.io/monaco-editor/`.
36. *Visual studio code* [online] [visited on 2022-05-10]. Available from: `https://code.visualstudio.com/`.
37. *Carpal tunnel syndrome* [online] [visited on 2022-05-10]. Available from: `https://bit.ly/39tsWhc`.