CENTRO UNIVERSITÁRIO CARIOCA
COMPUTER SCIENCE COURSE

PEDRO AUGUSTO RIBEIRO DA SILVA

VOICE-BASED ASSISTIVE TECHNOLOGY TOOL ABLE TO HELP PROGRAMMERS PROGRAMMING IN JAVASCRIPT LANGUAGE

RIO DE JANEIRO
2021

PEDRO AUGUSTO RIBEIRO DA SILVA

VOICE-BASED ASSISTIVE TECHNOLOGY TOOL ABLE TO HELP PROGRAMMERS PROGRAMMING IN JAVASCRIPT LANGUAGE

Course conclusion work presented to Centro Universitário Carioca (Unicarioca), as a partial requirement for obtaining a Bachelor's degree in Computer Science.

Advisor: Prof. Daisy Cristine Albuquerque da Silva, M.Sc.

RIO DE JANEIRO

2021

PEDRO AUGUSTO RIBEIRO DA SILVA

VOICE-BASED ASSISTIVE TECHNOLOGY TOOL ABLE TO HELP PROGRAMMERS PROGRAMMING IN JAVASCRIPT LANGUAGE

Approved by:

RIO DE JANEIRO
2021

THANKS

To God, for without him nothing is possible.

To my parents, for all their support and sacrifices for my benefit.

To my advisor, for all her help and patience during the development of this work.

To Unicarioca, its faculty and staff.

RESUME

Repetitive strain injury (RSI) is a syndrome that affects thousands of professionals around the world every year, among these professionals are programmers, professionals who tend to make extensive use of their hands in carrying out their tasks and that is why more susceptible to RSI. Programmers affected by this syndrome have great difficulty in using their hands and consequently great difficulty in carrying out their main activity: programming, a historically text-based activity that requires extensive use of the hands. That said, we come to the central theme of this work: the creation of an assistive technology tool capable of helping programmers to program in the JavaScript language using their voice. The tool aims to enable programmers to program without using their hands, using only voice commands.*parse* sentences belonging to a given grammar using stack automata, how to connect and control code editors using UI automation tools, the process of creating a code editor extension *Visual Studio Code* and the creation of a cross-platform application using ReactJS and the Electron framework.

Keywords: Speech recognition, stack automaton, assistive technology, user interface automation, speech-to-code conversion.

ABSTRACT

Repetitive strain injury (RSI) is a syndrome that affects thousands of professionals around the world every year, among these professionals are programmers, professionals who tend to make extensive use of their hands in carrying out tasks and that is why they are most susceptible to RSI Programmers affected by this syndrome have great difficulty in using their hands and entail great difficulty in carrying out their main activity: programming, a historically text-based activity that requires extensive use of the hands. That said, we come to the central theme of this work: the creation of an assistive technology tool capable of assisting programmers to program in JavaScript using their voice. The tool aims to enable programmers to program without the use of hands, using only voice commands.

Keywords: Speech-to-Text, Speech-to-code, speech recognition, stack automaton, assistive technology, user interface automation.

LIST OF ILLUSTRATIONS

Figure 1. Processes of a speech recognition system. Source: Silva (2009) 18

Figure 2. Simplified operation of a voice recognition service. Source: The author... 19

Figure 3. Example of the interaction between the client, Web Service and speech recognizer. Source: The author... 21

Figure 4. Using the SpeechSDK to perform the transcription of an audio file. Source: The author... 22

Figure 5. Cloud of the most common words in this document. Source: The author.........23

Figure 6. Simplified operation of a stop words removal function. Source: The author... 24

Figure 7. Automaton capable of recognizing some variations of the name Pedro. Source: The author... 26

Figure 8. Automaton with the sequence of states to go through to accept the “PIETRO” symbol chain in highlight. Source: Author... 27

Figure 9. Stack automaton represented as a finite state machine with access to a stack. Source: Adapted from Hopcroft; Motwani; Ullman, 1979................28

Figure 10. Automaton used to recognize the command to change lines. Source: The author... 30

Figure 11. Transition by similarity. Source: The author... .32

Figure 12. Example of using string distance in spell checker. Source: *print screen* from the website Google.com... 33

Figure 13. Automaton responsible for recognizing the command to write a number. Source: The author... 34

Figure 14. Showing how it is possible to make the composition between automata. Source: The author... 35

Figure 15. In the upper part of the image it is possible to see the text representation, using the DOT language, of the stack automaton and in the lower part the same automaton in visual format. Source: The author... 36

Figure 16. Diagram showing how the PyAutoGUI tool acts as an intermediary in communication between programs. Source: The author................................39

Figure 17. Diagram showing how a Visual Studio Code plugin can communicate with an external application. Source: The author...40
Figure 18. Spoken: The extension for VSCode proposed and developed in this project. Source:*print screen* of the Spoken extension.. 41
Figure 19. On the left the code in ReactJS used to create the initial screen of the application, on the right the result. Source: The author... 44
Figure 20. Application blueprint showing how all components interact with each other. Source: The author.. 46
Figure 21. Folder showing the structure of a command. Source: The author....................48
Figure 22. Contents of the “phrase_pt-BR.dot” file. On the left its textual representation and the visual right. Source: The author... 49
Figure 23. Contents of the “impl.js” file. Source: The author...50
Figure 24. Example of some methods that the extension must implement by contract. Source: The author.. 55
Figure 25. Implementation of write method using VSCode APIs. Source: The author... 55
Figure 26. Application home screen. Left in its natural state and right with expanded side menu. Source: print screen of the developed application.........57
Figure 27. Modules screen. Lists all application commands. On the left a list with all available commands, on the right details of the command to change lines. Source: *print screen* of the developed application..58
Figure 28. Example of the application in operation. On the left you can see the generated code and on the right the voice commands used. Source:*print screen* of the application developed interacting with the VSCode... 59
Figure 29. Application example at work. Source: application print screen.. 60

LIST OF TABLES

Table 1. Some of the interactions that the extension is required to be able to perform with the code editor............................... 42
Table 2. Some of the phrases recognized by the automaton in figure 21.....................50
Table 3. Some of the phrases recognized by the automaton in figure 21 and its

actions in the VSCode.. 51
Table 4. Commands implemented in the course of this project.................................53
Table 5. Set of stipulated basic functionalities.

LIST OF ABBREVIATIONS AND ACRONYMS

TA - Assistive Technology

READ - Repetitive Strain Injury

VSCode - Visual Studio Code STT -

Speech To Text

AST - Automatic Speech Recognition API -

Application Programming Interface SDK -

AFD Software Development Kit - Finite

Deterministic Automaton LIFO - Last-In-

First-Out

DOT - Language for graph representation in textual format GUI -

Graphical User Interface

IPC - Inter Process Communication

HTML - Hypertext Markup Language

CSS - Cascading Style Sheets

POST - One of the types of an HTTP request

SUMMARY

1. INTRODUCTION.. 13

1.1. GOALS.. .... .14

1.1.1. Main goal.. 14

1.1.2. Summary Objective.. 15

1.2. ORGANIZATION OF WORK 15

2. THEORETICAL FOUNDATION 16

2.1. VOICE COMMANDS RECOGNITION AND ANALYSIS....................................

2.1.1. SPEECH RECOGNITION 16

2.1.1.1. SPEECH RECOGNITION SOFTWARE................................18

2.1.1.1.1. AZURE SPEECH TO TEXT.. 19

2.1.1.1.1.1. SpeechSDK.. 21

2.1.1.1.1.2. Automatic Scoring.. 23

2.1.2. STOP WORDS.. 23

2.1.3. AUTOMATIC THEORY 25

2.1.3.1. Finite Deterministic Automaton 26

2.1.3.2. Stack Automaton.. 27

2.1.3.3. RECOGNIZING SENTENCES USING BATTERY AUTOMATS. .29

2.1.3.3.1. TRANSITION FUNCTIONS.. 31

2.1.3.3.1.1. TRANSITION BY SIMILARITY...32

2.1.3.3.1.2. Transition by Class.. 33

2.1.3.3.1.3. Transition by Automaton.. 34

2.1.3.4. Representing Automata in Textual Form......................................35

2.2. AUTOMATION OF CODE EDITORS.. 37

2.3.1. PyAutoGUI.. 37

2.3.2. EXTENSIONS.. 39

2.3.2.1 Spoken Extension For Visual Studio Code..40

2.4. JAVASCRIPT ECOSYSTEM.. 43

2.4.1. Electron Framework.. 43

2.4.2. React Framework.. 44

2.4.3. Express Framework.. 45

3. DEVELOPMENT OF THE SPEECH2CODE APPLICATION......................................45

3.1. VOICE COMMANDS 46
3.1.1. DEFINITION.. 47
3.1.1.1. DOT files... 48
3.1.1.2. Implementation File.. 50
3.1.2. List of Implemented Commands... 52
3.1.3. Transforming Voice Commands to Text..54
3.2. Spoken VSCode Extension... 54
3.3. USER INTERFACE 56
3.3.1. Home screen.. 56
3.3.2. Modules Screen.. 58
4. RESULTS... 59
4.1. Requirements for running the application... 62
5. CONCLUSION 62
BIBLIOGRAPHIC REFERENCES... 64

1. INTRODUCTION

Much has been said about the so-called accessibility software, software that aims to facilitate the use of computers and equipment by people with some kind of disability. These types of software fall within a larger area called Assistive Technology (AT). Assistive technologies are resources and services that aim to facilitate the development of activities of daily living for people with disabilities, promoting greater independence by allowing these people to perform tasks that they were previously unable or had great difficulty in performing. Assistive technologies help people who have difficulty speaking, typing, writing, seeing, hearing, learning, walking and many others. Different types of disabilities require different types of specialized assistive technologies. Some of the best known types of assistive technology include: the wheelchair, a device that facilitates locomotion for people who are unable or have difficulty walking; the hearing aid, a device that amplifies the sound waves of the environment in order to correct any hearing loss in the user; and so-called screen magnification software, software that allows you to magnify certain portions of the screen to help people with low vision.

In the context of diseases that affect the use of the hands, one of the best known is repetitive strain injury (RSI), a syndrome characterized by a group of diseases - tendonitis, tenosynovitis, carpal tunnel syndrome, trigger finger, etc. - that affects muscles, nerves and tendons of the upper limbs. In addition to causing pain and inflammation, this disorder can also compromise the affected region. RSI is usually caused by repetitive movements such as typing, poor posture, playing the piano, driving, crochet and so on.

According to data from the Social Security Statistical Yearbook, in 2019, LER was among the 50 most common causes of work accidents in Brazil. Among these causes are tenosynovitis (inflammation or infection in the tissue covering the tendon) and mononeuropathy of the upper limbs (injury to the peripheral nerve), both conditions that compromise the use of the hands.

Individuals unable or with difficulty to interact with computers due to RSI can count on the help of assistive technologies based on voice recognition, in this type of technology the user uses their voice, instead hands, to interact directly with the computer. Some examples of assistive technologies based on voice recognition include voice assistants: Siri, Alexa and Cortana. Using Siri's voice assistant, for example, it is possible to write and send an email without using your hands, using only your voice.

Among the countless professionals affected by RSI are programmers, professionals whose work demands the constant use of their hands, and therefore, more susceptible to RSI. This is due to the fact that, historically, the software development process has taken place through repeated interactions between mouse, keyboard and hands. An example of this are code editors, where the main form of interaction is through the keyboard (BEGEL, 2005).

That said, we come to the central theme of this work: the creation of an assistive technology tool based on voice recognition capable of helping programmers with RSI to program.

1.1. GOALS

The main objectives to be achieved in the development of this work will be listed below.

1.1.1. Main goal

The objective of this work is to create an application capable of helping programmers to program in JavaScript without using their hands. This application will be able to listen and analyze voice commands and then manipulate any code editor in order to execute this command.

Using this application instead of using your hands to write in the code editor, for example, the instructions needed to declare a variable, the programmer can simply express aloud that he wants to declare a variable which then the application will automatically connect to the code editor with the intention of writing the necessary instructions to declare a variable in the JavaScript language.

For this, the application must be able to: capture voice commands through from the computer's microphone, transform the voice command into text, validate the command and, finally, connect to the code editor in order to execute the command.

1.1.2. Summary Objective

- Implement a functionality capable of real-time voice recognition. This functionality will turn voice commands into text.

- Define a language and develop a mechanism to recognize that language using finite automata. The intent of this language is to represent all available voice commands in textual form.

- Develop a tool capable of interacting with code editors. This tool will be able to manipulate a code editor to, for example, remove a line.

1.2. WORK ORGANIZATION

This work is organized into 5 fundamental chapters that fully expose its content. The chapters are:

- **Chapter 1 - Introduction:**
  This chapter. It aims to briefly present the central theme, motivation, organization and objectives of this work.

- **Chapter 2 – Theoretical Foundation:**
  In this chapter, the tools and concepts used during the development of the work will be presented. The concepts of finite deterministic automaton will be presented,*STT Services (speech-to-text services), string distance,* UI automation and so on.

- **Chapter 3 – Speech2Code Application Development:**

This chapter will show and document how all the concepts and technologies described in chapter 2 interact with each other to achieve the proposed objectives of this work.

- **Chapter 4 - Results:**
  In addition to a step-by-step demonstration of how to use the application, this chapter will also assess which sets of basic JavaScript programming language commands have been implemented.

- **Chapter 5 - Conclusion:**
  This chapter presents a summary of the work, its objectives, the degree to which these objectives were achieved and suggestions for future improvements.

2. THEORETICAL FOUNDATION

Next, the concepts and technologies used throughout the development of this project will be presented.

2.1. VOICE COMMANDS RECOGNITION AND ANALYSIS

A command is a phrase that is directly associated with an action that involves manipulating the code editor. For example, the phrase “Go to line 2” can be characterized as a command to change lines whose action is to manipulate the code editor so that the line is changed. The application of assistive technology proposed in this work works by listening to user voice commands and then performing actions associated with these commands.

Next, all the concepts and technologies involved in the task of transforming a sentence said aloud into text will be listed, analyzing this text in order to characterize it as a valid command and extract important information for carrying out such a command.

2.1.1. SPEECH RECOGNITION

Speech recognition, also known as automatic speech recognition, is an interdisciplinary area of computational linguistics that aims to develop methods and technologies that allow computers to transform spoken language into textual language. Although the term voice recognition is often used as a synonym for speech recognition, the latter focuses on transforming spoken language into text, while the former focuses on identifying individual people's voices (IBM Cloud, 2020).

In addition to the peculiarities of human language, problems of an interdisciplinary nature make speech recognition one of the most difficult areas in computer science. Among the disciplines involved in solving problems in this area are (RABINER, Lawrence; JUANG Biing-Hwang, 1993):

- Signal processing: The process of extracting relevant information from the acoustic signal generated by the voice efficiently and robustly.

- Physics (Acoustics): The science of the relationship between the physical manifestation of speech (signal generated by the voice), the physiological mechanism that produces speech (the human vocal system) and how speech is perceived (the human auditory system).

- Pattern Recognition: The set of algorithms used to identify and classify clustering patterns in a database.

- Linguistics: The relationship between sounds (phonology), words in a language (syntax), meaning of words (semantics) and meaning derived from context (pragmatics).

A speech recognizer has as input a speech signal obtained from a transducer and from this signal a mapping is carried out in order to discover the spoken word, that is, to transcribe what was spoken. The recognition process is divided into four steps, voice signal acquisition, pre-processing, information extraction and the last one, which can be the generation of voice patterns, when in the training phase or classification, when in the recognition phase , what uses the voice patterns generated in the training phase (Silva, 2009).

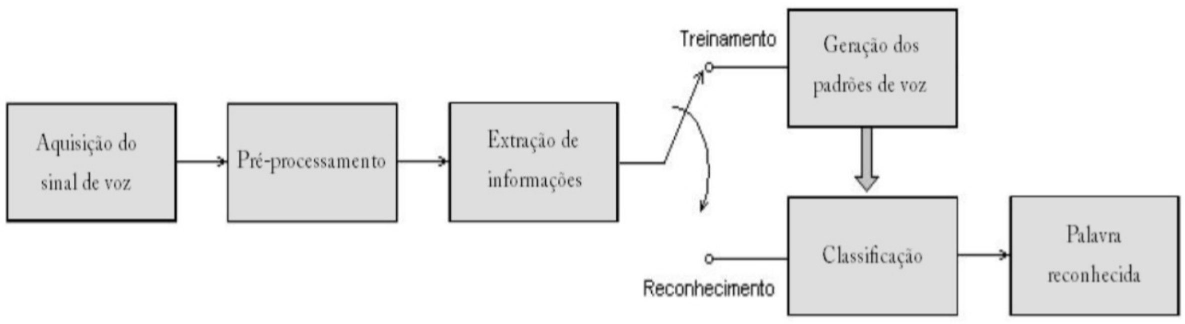


Figure 1. Processes of a speech recognition system. Source: Silva (2009)

A speech recognizer is evaluated based on its accuracy rate, that is, word recognition error rate and recognition speed. Various factors can affect the error rate, such as pronunciation, accent, tone, volume and noise. Speech recognizers aim to achieve an error rate similar to that of two humans talking (IBM Cloud, 2020).

Speech recognition systems have applications in several areas. In fact, any activity that involves human-machine interaction can potentially utilize these systems. Currently, several applications are already being designed with a built-in speech recognition system. Among the most common applications are: control and command systems, telephone systems, transcription systems, customer service centers, virtual assistants and security (Silva, 2009).

2.1.1.1. SPEECH RECOGNITION SOFTWARE

In few words, *speech to text software (STT),* or *automatic speech recognition software (ASR),* or *voice to text software,* is a computer program that uses linguistic algorithms to classify and transform sound signals into text.

Voice recognition software can be used in cases where the individual needs to generate a large volume of textual content without the need of writing manually. This technology is also used by individuals who are unable or have great difficulty using a keyboard.

Among the various products on the market for voice recognition, the following stand out: *Cloud Speech-to-text (*Google), *Azure Speech to Text (*Microsoft) and the *Watson Speech to Text (*IBM), all are cloud-based services capable of multi-lingual speech recognition and promise high accuracy in word recognition. As they are products aimed at general use, not limited to a specific domain, they tend to be easy to use and low cost.

To carry out this work, the voice recognition tools from Google and Microsoft were considered. Although both work in a similar way, through web APIs, and support the same functionalities, the Microsoft tool presented a higher degree of accuracy in word recognition and has a more comprehensive documentation. That said, in this project we will use as a speech recognition tool the*Azure Speech To Text.*

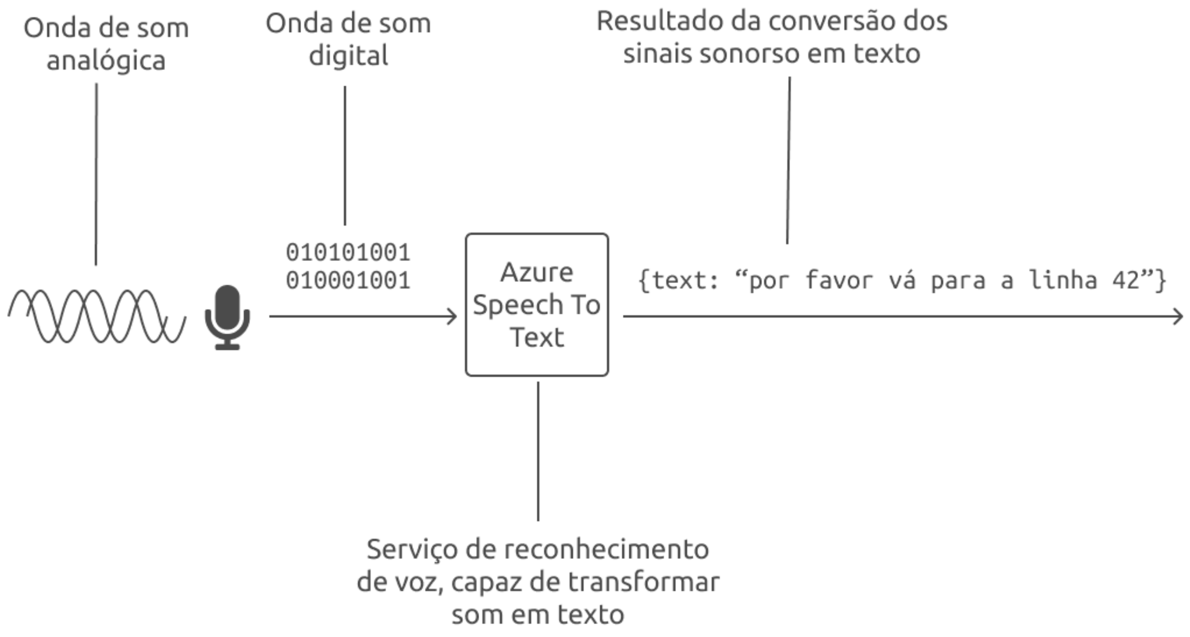

Onda de som analógica
Onda de som digital
Resultado da conversão dos sinais sonorso em texto
010101001
010001001
Azure Speech To Text
{text: "por favor vá para a linha 42"}
Serviço de reconhecimento de voz, capaz de transformar som em texto

Figure 2. Simplified operation of a voice recognition service. Source: The author

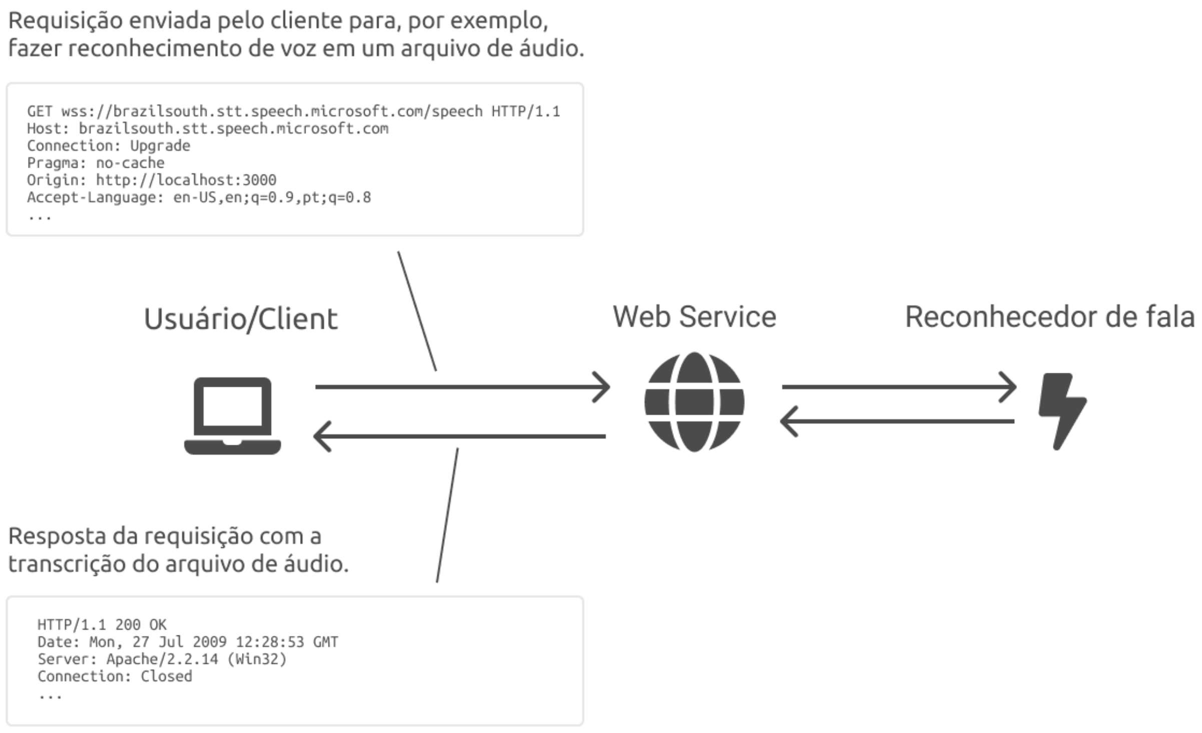
2.1.1.1.1. AZURE SPEECH TO TEXT

O *Microsoft Azure Speech To Text,* commercial recognition software

Microsoft voice allows you to create applications with speech recognition capabilities simply and quickly. It has support for over 85 different languages, high accuracy rate, support for different programming languages   and template customization. It also offers audio to text conversion from different sources like microphones, audio files and
blobs.

Because it is a *SaaS (Software as Service)* The first thing to do before using the service is to create an account on Microsoft Azure, Microsoft's cloud computing platform. With this account, it is possible to hire a wide range of services and products aimed at different topics such as data storage, security, internet of things, artificial intelligence, and so on. Azure offers several different account types for different profiles, ranging from the most basic to the most robust, including a free account. In the context of this project, one of the benefits of the free account is 5 hours per month of free speech-to-text conversion, that is, it is possible to carry out up to 5 hours of speech recognition per month without paying anything for it.

The use of *Azure Speech To Text* through a Web Service available to users with a Microsoft Azure account. This Web Service acts as an intermediary for the speech recognition feature. Its function is to receive and validate requests for speech recognition and then respond accordingly. Using this Web Service, it is possible, for example, to send a request to perform speech recognition in an audio file and receive, in response, the text transcription of this file.


Requisição enviada pelo cliente para, por exemplo, fazer reconhecimento de voz em um arquivo de áudio.
Usuário/Client
Web Service
Reconhecedor de fala
Resposta da requisição com a transcrição do arquivo de áudio.

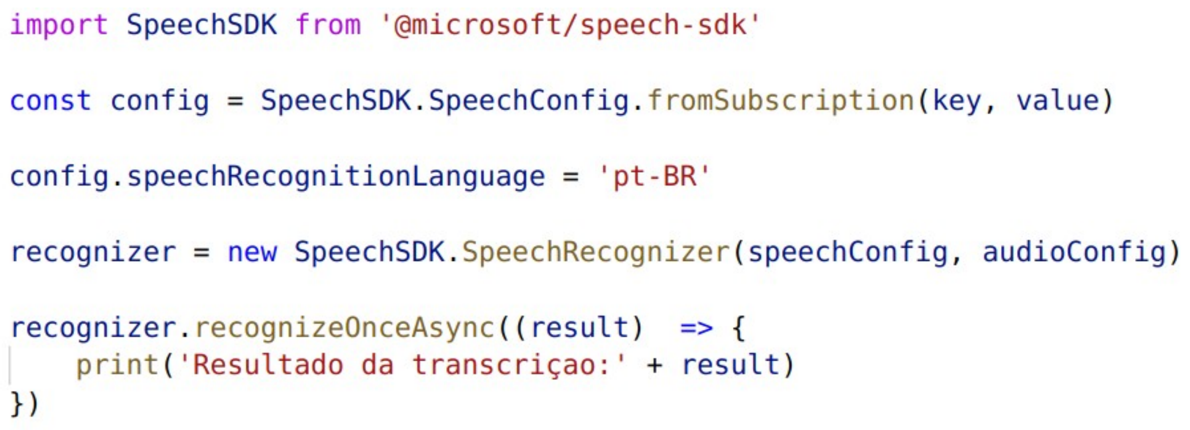
```
GET wss://brazilsouth.stt.speech.microsoft.com/speech HTTP/1.1
Host: brazilsouth.stt.speech.microsoft.com
Connection: Upgrade
Pragma: no-cache
Origin: http://localhost:3000
Accept-Language: en-US,en;q=0.9,pt;q=0.8
...
```

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Connection: Closed
...
```

Figure 3. Example of the interaction between the client, Web Service and speech recognizer. Source: The author

2.1.1.1.1.1. speech SDK

Azure offers a *SDK (*Software Development Kit) called *speech SDK* which aims to facilitate interaction with the Web Service, this SDK abstracts and encapsulates all the complexity of the communication process with the Web Service. The SDK is available in several programming languages including Java, JavaScript, Python and so on. In this project, its implementation in JavaScript was used, as it is a project aimed at the Web. Using the SDK, it is possible to carry out the process described in Figure 3 in a few lines of code as shown below:



```
import SpeechSDK from '@microsoft/speech-sdk'

const config = SpeechSDK.SpeechConfig.fromSubscription(key, value)

config.speechRecognitionLanguage = 'pt-BR'

recognizer = new SpeechSDK.SpeechRecognizer(speechConfig, audioConfig)

recognizer.recognizeOnceAsync((result)  => {
    print('Resultado da transcriçao:' + result)
})
```

Figure 4. Using the SpeechSDK to perform the transcription of an audio file. Source: The author

The SDK offers 3 basic types of speech recognition all of which can be done from a microphone, file or *blob:*

- Spot Recognition - Recognition is done throughout speech until the user stops talking. “Stop talking” is considered to be a long pause in speech. This type of speech recognition is indicated for commands and questions, cases in which after the user stops speaking, the recognition process should end automatically.

- Continuous Recognition - Recognition is done continuously throughout the speech until the user explicitly stops it. It is indicated in cases where the recognition process must continue even in long periods of silence. Examples of usage include monologues and conversations between two or more people.

- Keyword Recognition - Recognition only starts when the user says a keyword and only ends when the user explicitly stops it. In this type of recognition speech is always recorded, analyzed and the results discarded, unless the keyword is said, then the results are not discarded but rather delivered to the user. Examples of this type of speech recognition include the “Ok Google, ...” command (where “Ok Google” is the keyword) from the voice assistant *Google Assistant.*

2.1.1.1.1.2. Automatic scoring

Another important feature of the *Azure Speech To Text* for the context of this project it is automatic scoring in sentence recognition. This functionality is able to analyze a spoken sentence and infer where, according to the grammar rules of the language, punctuation marks are needed. With this feature activated by saying the phrase “who are you” in a question tone and then performing the speech recognition process the result would be the phrase “who are you?” with the question mark sign.

When disabling this functionality, the inclusion of punctuation in the speech transcription must be done explicitly. When saying the phrase "Yellow, Blue and Green" the result will not be "Yellow, Blue and Green", to achieve such a result it is necessary to say "Yellow Comma Blue and Green", that is, the comma - punctuation - must be explicit in the sentence . This difference is of utmost importance for this project since punctuation marks have an important meaning in programming.

2.1.2. STOP WORDS

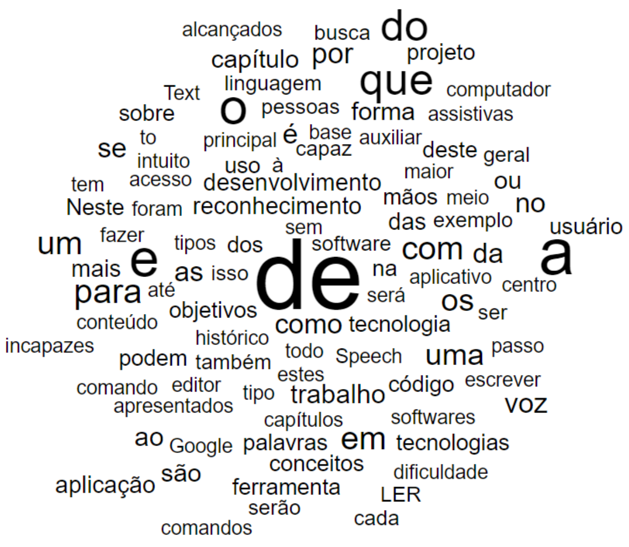


Figure 5. Most common word cloud for this document. Source: The author

In the context of natural language processing, *stop words* or stop words are words that can be removed before or after processing documents. *Stop words* they are usually classified as the most common words of a language, with high chances of occurring in any document, and they tend to contribute more to the grammatical and syntactic conformity of the language than to the actual meaning of the document (Wilbur; Sirotkin,
1992). There is no defined list of which words are classified as*stop words* and they can vary from application to application.

In the sentence "Go to line number 42" the words "to" and "a" can be classified as *stop words* and removed without significant loss in sentence sense, this is because the words “to” and “a” exist only to complete the indirect verb “go”.

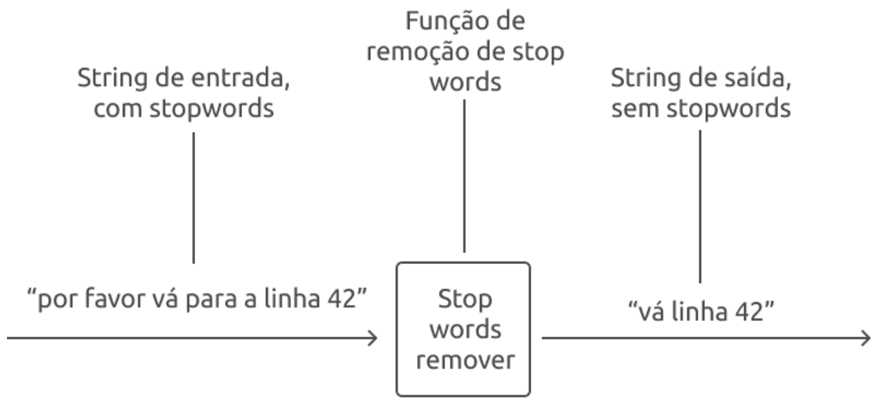

Função de remoção de stop words
String de entrada, com stopwords
String de saída, sem stopwords
“por favor vá para a linha 42”
Stop words remover
“vá linha 42”

Figure 6. Simplified operation of a removal function
stop words. Source: The author

In this project *stopwords* will be removed to simplify the command recognition process. Imagine that the user said a phrase aloud and, after doing speech recognition, it was found that he said the phrase “please declare a variable called value equal to the number 42”. The next phase would be to check if this sentence corresponds to a valid command or not, but as seen previously*stopwords* these are words that do not influence the meaning of the sentence, so we can remove them. Therefore, the sentence sent to the next stage would be "please declare a va~~riable call~~ed value ~~equa~~l to the number 42", effectively re~~m~~oving the stop words "please", "one" and "ao" from the sentence original.

2.1.3. AUTOMATIC THEORY

Automata Theory is a theoretical branch of computer science that has its roots in the 20th century, when mathematicians began to develop both in theory and in practice - machines that mimicked certain functions of the human brain (Hopcroft; Motwani; Ullman, 1979). The word automaton itself, closely related to the word "automation", denotes automatic processes that carry out a series of pre-determined operations automatically. Simply put, automata theory deals with the computation logic of simple machines, called automata. Through automata, computer scientists are able to understand how machines perform functions and solve problems and, more importantly, what it means for a function to be computable or a decidable question.

Automata are abstract models of machines that analyze a string of characters moving through a series of states. In each state, a transition function determines the next state based on a part of the input string. As a result, once the automaton reaches an accepting state, it is said to accept that input.

The main objective of automata theory is to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems. The behavior of these discrete systems is determined by how the system is built from storage and combinational elements (Eric Roberts, 2009). Features of such machines include:

- Input: sequence of symbols selected from a finite set of input signals.
- Output: sequence of selected symbols from a finite set M. Where M is the set {a1, b2, b3 ... bn} where n is the total number of outputs.
- States: finite set Q, whose definition depends on the type of automaton.

There are three main families of automata: finite state machine, stack automata, and Turing machines. These automata families can be interpreted in a hierarchical way, where the finite state machine is the simplest automaton and the Turing machine is the most complex. The focus of this project is on the finite state machine and the stack automaton.

2.1.3.1. Finite Deterministic Automaton

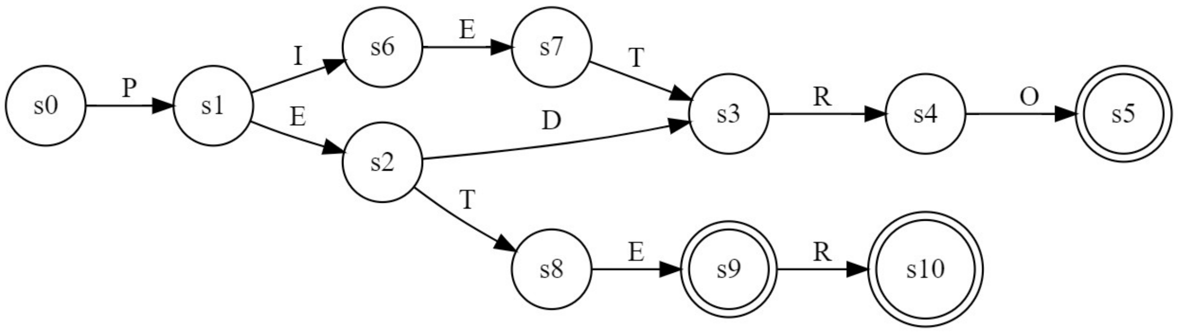


Figure 7. Automaton capable of recognizing some variations of the name Pedro. Source: The author

Finite deterministic automaton (AFD) or finite deterministic state machine is a state machine that, when traversing a unique sequence of states, accepts or rejects a chain of symbols (Hopcroft; Motwani; Ullman, 1979). In the automaton in the figure above, the states are represented by circles, the final states by double circles, the transition functions by arrows and the initial state by*s0.*
An automaton is said to accept a string of symbols, if when consuming all the symbols of that string it finds itself in a final state, otherwise it rejects it.

Formally a Finite Deterministic Automaton A is a quintuplet, (Q, Σ, δ, q0, F) where:

- Σ is the finite set of symbols called the alphabet
- Q is the finite set of states
- δ is the transition function
- q0 is the initial state (q0 ∈ Q)

● F is the set of acceptance states (F ⊆ Q)

In the theory of automata a finite state machine is only considered a finite deterministic automaton if:

● Each of your transitions is determined only by the source state and input symbol.
● It is necessary to read an input symbol for a change of state.

If the finite state machine does not meet these conditions, it is called a non-deterministic finite automaton.

The AFD shown in the figure below recognizes the chain of symbols "PIETRO" when scrolling through the states: *s0, s1, s6, s7, s3, s4* and *s5,* being *only* and *s5* initial and final states, respectively.

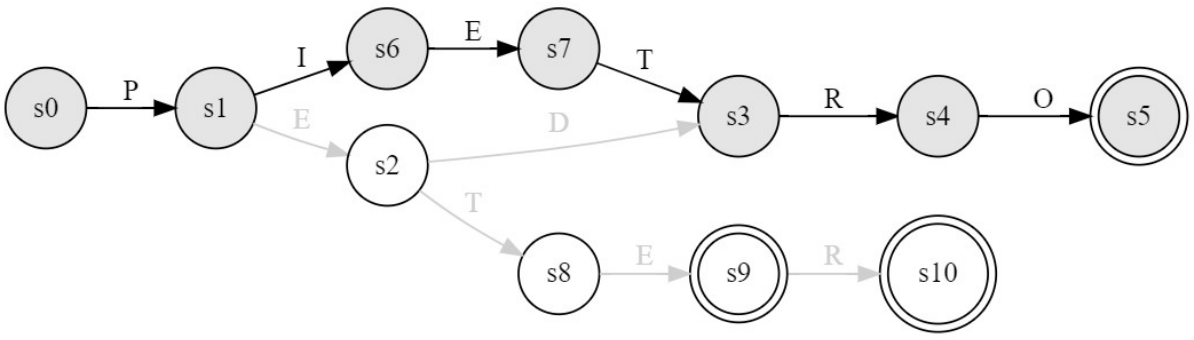


Figure 8. Automaton with the sequence of states to go through to accept the symbol string "PIETRO" highlighted. Source: Author

AFDs can be used in solving problems involving regular languages such as regular expressions and in lexical analysis of a compiler. For example, it is possible to model an automaton capable of telling whether an email address is valid or not (GOUDA; Prabhakar, 2009).

2.1.3.2. cell automaton

The stack automaton is essentially a non-deterministic finite automaton, which accepts empty transitions (λ), with the addition of one more feature: a stack in which it can store a series of symbols. The presence of this stack means that, unlike the finite deterministic automaton, the stack automaton can hold an infinite amount of information. However, unlike a general-purpose computer, which can also store a large amount of information, the stack automaton can only access the information in its memory/stack in one model.*last-in-first-out (LIFO),* characteristic of the stack data structure.

Due to this limitation there are languages that can be recognized by a general purpose computer, but not by stack automata. Stack automata by definition only recognize context-free languages (Hopcroft; Motwani; Ullman, 1979). According to Chomsky's Hierarchy, stack automata are computation models equivalent to context-free grammars.

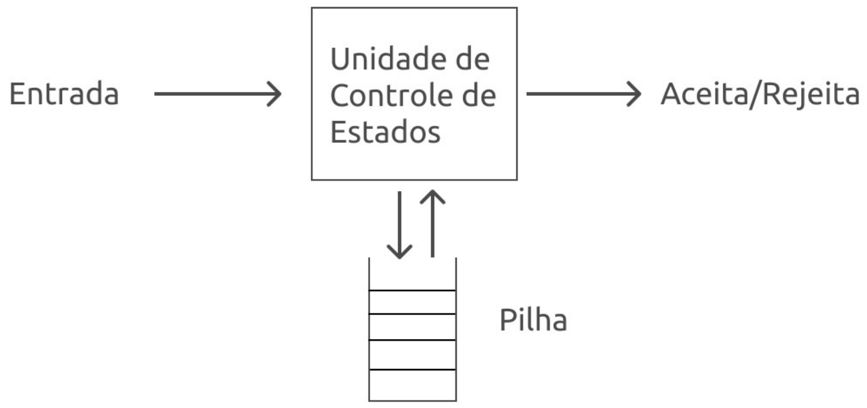


Figure 9. Stack automaton represented as a finite state machine with access to a stack. Source: Adapted from Hopcroft; Motwani; Ullman, 1979.

The device shown in the figure above exemplifies the operation of a battery automaton. As the “state control unit” reads the input symbols, the stack automaton can base its change of state on its current state, the symbol at the top of the stack and the symbol read. The change of state can even occur spontaneously without reading a symbol from the input, this spontaneous transition is called empty transition or transition*epsilon (λ).*

According to Hopcroft, Motwani and Ullman (1979) in each transition an automaton of battery:

1. Read a symbol from the input that will be used to make the transition, if it is an empty transition no symbol is read (*λ).*

2. Go to a new state which can be the same as the previous state.

3. Replace the symbol at the top of the stack with another symbol. This other symbol can be an empty string (λ), which corresponds to the removal of the symbol at the top of the stack. The same symbol that is already at the top of the stack, which corresponds to no change in the stack. Any other symbol, which corresponds to the replacement of the top of the stack, without removal or addition. And finally the top of the stack can be replaced by two or more symbols, which can have the effect of replacing the top of the stack and then adding more symbols to it.

2.1.3.3. RECOGNIZING SENTENCES USING BATTERY AUTOMATS

As we saw earlier, a command is a phrase associated with an action that involves manipulating a code editor. Given a sentence in textual format, the problem of deciding whether or not that sentence is a valid command can be solved with the aid of stack automata where the input alphabet and transition functions are based on words. The idea is that a stack automaton, capable of recognizing one or more sentences, is associated with a command. Consider the command A that is associated with automaton B that recognizes a set of sentences K, also consider any phrase x. The action related to command A will only be performed if sentence x belongs to K.

When using a stack automaton to recognize a sentence, the input symbols that make up the alphabet can be treated as the individual words that make up the sentence. An A-stack automaton recognizes the phrase “go to line 42” if when it runs through all the words in that sentence (go, to, to, line, 42) it finds itself in a final state. The extraction of the words that make up the sentence is done using a *tokenizer* simple.

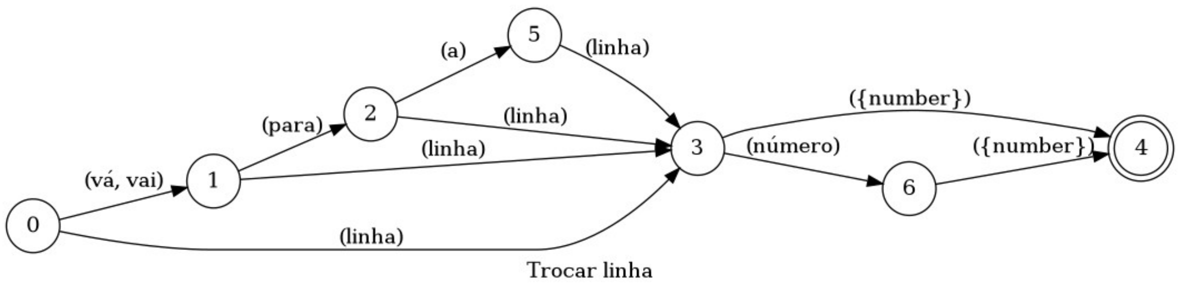


Figure 10. Automaton used to recognize the command to change lines. Source: The author

The stack automaton in the figure above is responsible for recognizing phrases related to the new line command. As stated before, all transition functions are based on words and not symbols, yet in Figure 10 we can see that the transition from state 0 to state 1 only occurs if the input word is "go" or "go", the transition from state 1 to state 2 only when the input word equals “to” and so on. Another peculiarity is the notation "({*number})*" used in the transition between states 3 and 4, this notation denotes that the transition must accept any read word and store it at the top of the stack.

Altogether, the stack automaton in figure 10 can recognize 8 sentences related to the line change command, below we will see how it behaves when trying to recognize the sentence "go to line 42" that after going through the *tokenizer* results in the input string {"go", "to", "line", "42"}:

1. In initial state 0, the first word of the input, “go”, is read. State 0 has transitions for states 3 and 1, but only the transition function for state 1 accepts the word “go”, so the automaton consumes the input word and goes to state 1.

2. In state 1, the next word in the input sequence, “to” is read. State 1 has transitions for states 2 and 3, but only the transition function for state 2 accepts the word “para”, so the automaton consumes the input word and goes to state 2.

3. In state 2, the next word in the input sequence, “line”, is read. State 2 has transitions to states 3 and 5, but only the transition function to state 3 accepts the word "line", so the automaton consumes the input word and goes to state 3.

4. In state 3, the next word in the input sequence, “42” is read. State 3 has transitions to states 4 and 6. The transition function to state 6 requires the word “number” which is different from the word “42”. The transition function to state 4, on the other hand, accepts any word with the condition that it is written to the stack. In this case the automaton consumes the input word and adds it to the top of the stack, finally it goes to state 4.

5. In state 4 there are no more words in the input sequence and the automaton is in a final state. It is then said that the automaton identifies the phrase "go to line 42" as being a valid command to change lines and its stack, which currently contains the word "42", can be used as an argument when performing the action related to that command.

As shown, the automaton in figure 6 recognizes the phrase “go to line 42” as a valid phrase for the new line command, it goes through states 0, 1, 2, 3, 4 and ends up with the stack containing the word “42”. The elements of this stack can be used as arguments to execute the action associated with the command, in the case presented, the action is to change lines and the argument is line number 42.

That said, given an M set of commands, deciding whether a phrase x is a valid command or not is just a matter of iterating over each element of M and using its associated automaton to decide whether or not the phrase belongs to the command.

2.1.3.3.1. TRANSITION FUNCTIONS

Another significant change made to the stack automata in order to decreasing its complexity was the definition of 3 types of transition function. Recall that the transition function in a stack automaton decides whether the automaton can go to the next state based on the input symbol, the current state, and the top of the stack (Hopcroft; Motwani; Ullman, 1979).

2.1.3.3.1.1. TRANSITION BY SIMILARITY

Normally the transition function compares the symbol read, with the symbol needed to go to the next state, if they are equal the transition is made. In the transition by similarity, this process is changed so that the transition is also made if the symbols are similar.

Using this type of transition, it is possible to filter out verbal variations in words and mistakes made by the voice recognition process. Consider the automaton below, it accepts both the phrases “declare one” and “declare one”, this is due to the fact that although the words “declare” and “declare” are different they are still very similar.

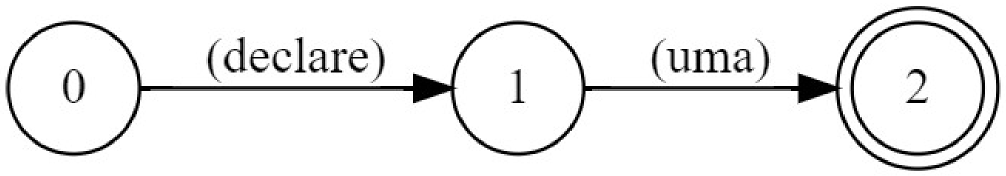


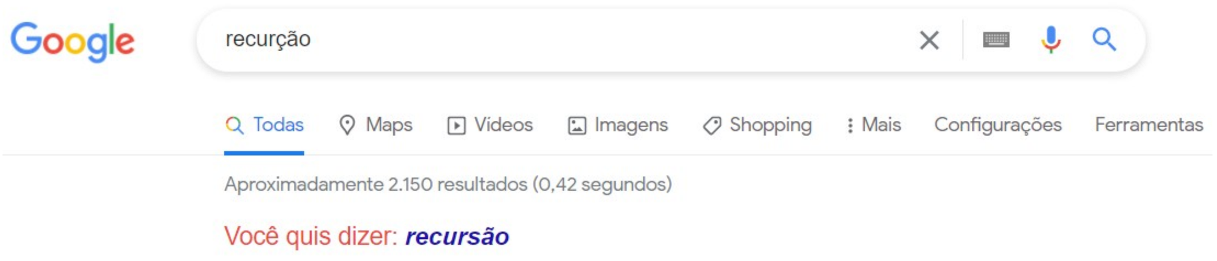
Figure 11. Transition by similarity. Source: The author

Similarity transition is the default and is recommended for all transitions involving words.

The similarity between two words is determined through an algorithm that measures the distance between two strings (character string also called a word).

2.1.3.3.1.1.1. String Distance

string distance or *string metric* is a metric that measures the degree of similarity between two text fragments, this metric is widely used in problems such as approximate text search and spell check. For example, the words “Emerge” and “Immerse” can be considered similar (Lu; et al, 2013).



Aproximadamente 2.150 resultados (0,42 segundos)

Você quis dizer: ***recursão***

Figure 12. Example of using string distance in spell checker. Source:*print screen* of website Google.com

Among the various algorithms to calculate the similarity between *strings* (strings) the most popular is the so-called Levenshtein distance, which measures the number of edits (insertion, deletion or replacement) to transform one string into another. For example, the distance Levenshtein between the English words "kitten" (cat) and "sitting" (sitting) is 3, since with just 3 edits we can transform one word into another.

In this project the metric *string distance* will be used as the basis for the transition function of the automatons responsible for recognizing and analyzing commands. The use of this metric makes the automaton more tolerant to spelling errors and verbal variations.

2.1.3.3.1.2. Transition by Class

The transition is made when the input word belongs to a predetermined class of words. This transition is used in cases where it is not possible to enumerate all the words accepted by a transition, so they are grouped under class X and the transition is said to occur for any word that belongs to group X. Class transitions exist with intent to simplify the representation and share logic between different automata.

The automaton in the figure below is able to recognize the phrases "number 1",

“number 2”, “number 3”, “number 4” and so on. It is able to recognize any phrase in the format “number $*X*” where $*X* is any natural number. This is because the transition from state 2 to 3 is a class transition, so this transition will go to the next state when reading any word that belongs to the class.
{number}.

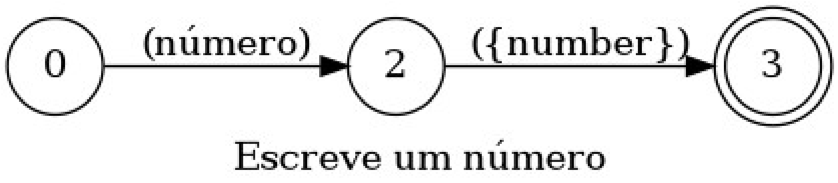


Figure 13. Automaton responsible for recognizing the command to write a number. Source: The author

Internally the class {*number}* is defined as a regular expression that accepts any natural number. Other classes also include: {*char},* which accepts only one character, {*term},* that accepts any word and {*numeral}* which accepts ordinary numbers.

Class transitions must always be denoted between curly braces otherwise they will be interpreted as a similarity transition. In figure 13 the transition from state 0 to state 2 is a transition of similarity while the transition from state 2 to 3 is a class transition, this is due to the fact that the text of the latter is between the braces. .

2.1.3.3.1.3. Transition by Automaton

In this type of transition, control over the input sequence is passed to a second automaton and the state transition only occurs if this second automaton ends in a final state. Essentially this type of transition allows the composition and reuse of automatons.

In the figure below, it is possible to observe that automata B and C use automaton A to compose their transition functions. Red arrows indicate which automaton the transition function references.

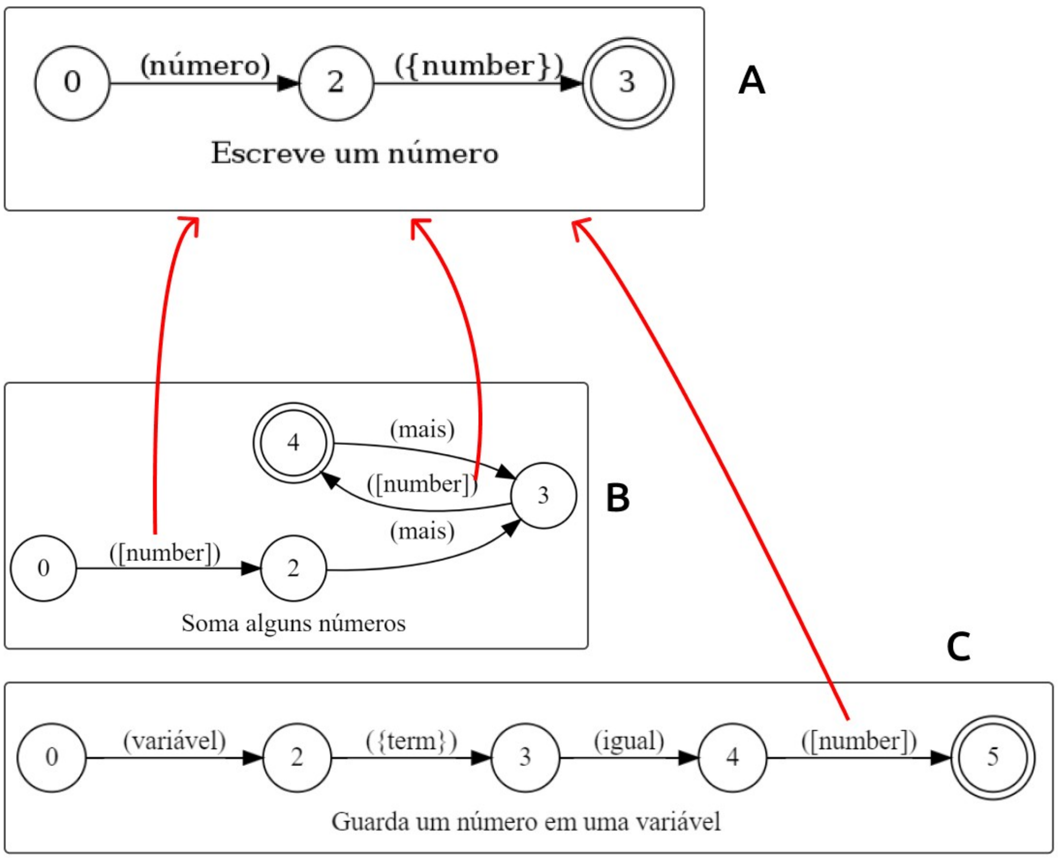


Figure 14. Showing how it is possible to make the composition between automata. Source: The author

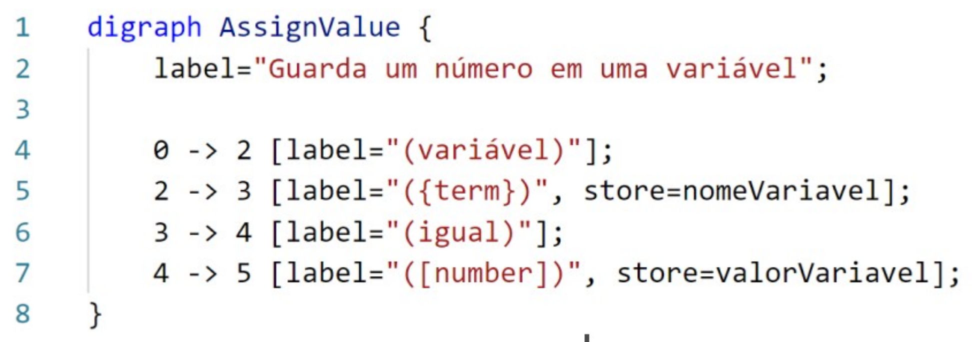
In automaton B to make the transition between states 0 and 2, the current content of the input sequence is applied to automaton A, if it ends in a final state, automaton B can proceed to state 2. It is worth noting that automaton A it can also consume input words and manipulate the stack.

Automaton transitions must always be denoted in square brackets, otherwise they will be interpreted as a similarity transition. In automaton C in figure 14, the transition from state 0 to state 2 is a transition of similarity while the transition from state 4 to 5 is a transition of automaton, this is due to the fact that the text of the latter is between the brackets.

2.1.3.4. Representing Automata in Textual Form

DOT is a language for describing graphs in textual form which can then be turned into diagrams. Using the *DOT* it is possible to represent directed graphs, undirected graphs and even automata (Gansner; Koutsofios; North, 2015).

As mentioned earlier, stack automata will be an important part of this project and the language. *DOT* presented here will be the standard way of representing them.



```
digraph AssignValue {
    label="Guarda um número em uma variável";

    0 -> 2 [label="(variável)"];
    2 -> 3 [label="({term})", store=nomeVariavel];
    3 -> 4 [label="(igual)"];
    4 -> 5 [label="([number])", store=valorVariavel];
}
```

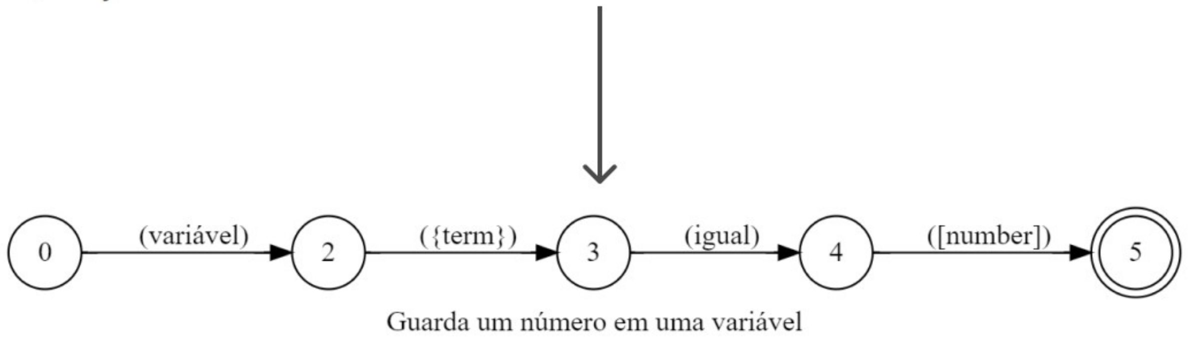


Figure 15. In the upper part of the image it is possible to see the text representation, using the DOT language, of the stack automaton and in the lower part the same automaton in visual format.

Source: The author

In Figure 15, you can see the three types of transitions already mentioned:

- 0 → 2: Transition by similarity, accepts any word similar to the word “variable”.

- 2 → 3: Transition by class, accepts any word belonging to the “term” class.

4 → 5: Transition by automaton, applies the current input to an automaton called “number” and only goes to the next state if this automaton accepts it.

Still in the textual representation of Figure 15, it is possible to observe that the transitions in lines 6 and 7 have an attribute called "*store",* this attribute tells the automaton to store the read word at the top of the queue.

2.2. AUTOMATION OF CODE EDITORS

As explained above, the main objective of this work is to create an assistive technology tool capable of helping programmers to program. The tool eliminates the need for the programmer to use the keyboard to interact with the code editor, the programmer just says aloud what needs to be done and this will be done automatically in the code editor. This functionality can be divided into two major problems: recognition and analysis of the voice command and manipulation of the code editor to execute the command. In this subchapter we will explore the second problem: How a program A can interact with another external program B.

GUI Automation or Graphical User Interface Automation is the process of programmatically simulating mouse and keyboard actions in order to interact with an external program, can be used in screen readers, automated testing, automated data entry, program integration, and migration of content. Generally,*GUI Automation* it is used to automate repetitive and tedious tasks (UIPATH, 2015).

In this project we will use two ways to *GUI Automation* to interact with code editors: *PyAutoGUI* and Extensions.

2.3.1. PyAutoGUI

PyAutoGUI is a Python programming language package that allows mouse and keyboard control to automate interactions with other programs

(Sweigart, 2017).

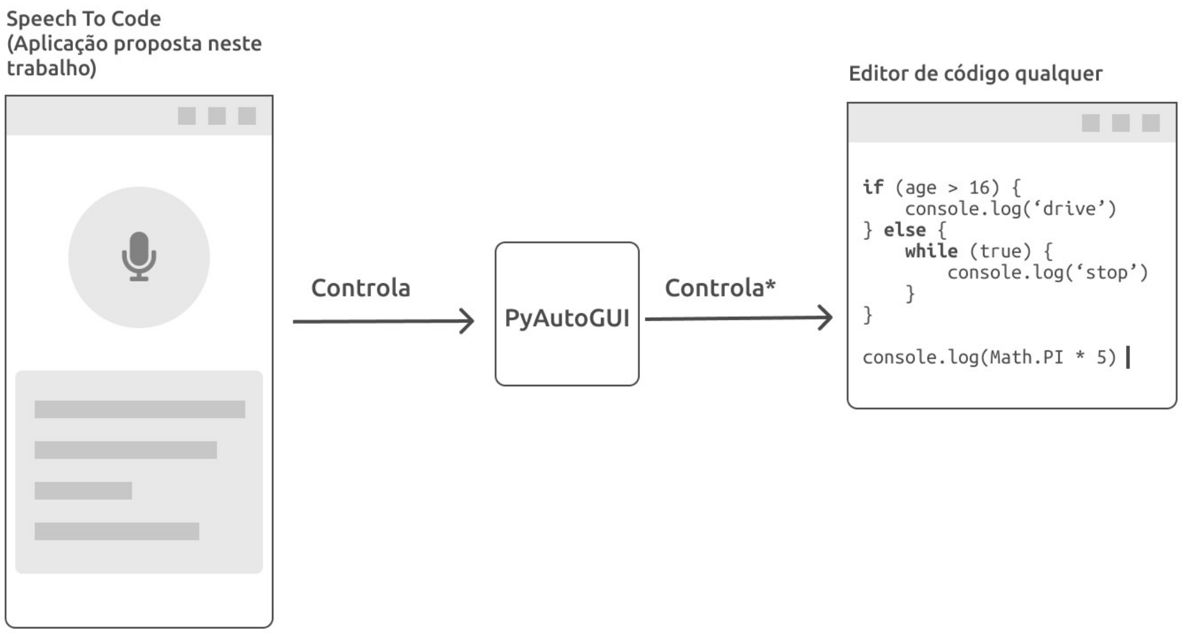
Using this package it is possible:

- Move and click with the mouse or write in windows of other applications.
- Control the keyboard to, for example, fill in forms.
- Find the coordinates of elements such as buttons and images on the screen.
- Close, move, resize or maximize windows from other applications.

One of the biggest advantages of PyAutoGUI is being able to interact with any application, it is possible, for example, to use the same script to write in the Whatsapp message box to fill the application forms of the Income Tax Declaration. This is due to the fact that for PyAutoGUI what is on the screen is not the X or Y application but a series of pixels.

In the context of this project this generalization becomes somewhat problematic, for example, using PyAutoGUI to interact with code editors simple tasks like changing the pointer to an arbitrary line or indenting the current file become complicated, since PyAutoGUI does not have the concept of code editor or number of lines in a file, but pixels on a screen.

That said, PyAutoGUI will not be the default automation tool and will only be used in cases where a more expert alternative is not available.


Speech To Code (Aplicação proposta neste trabalho)
Controla
PyAutoGUI
Controla*
Editor de código qualquer

```
if (age > 16) {
    console.log('drive')
} else {
    while (true) {
        console.log('stop')
    }
}

console.log(Math.PI * 5)
```

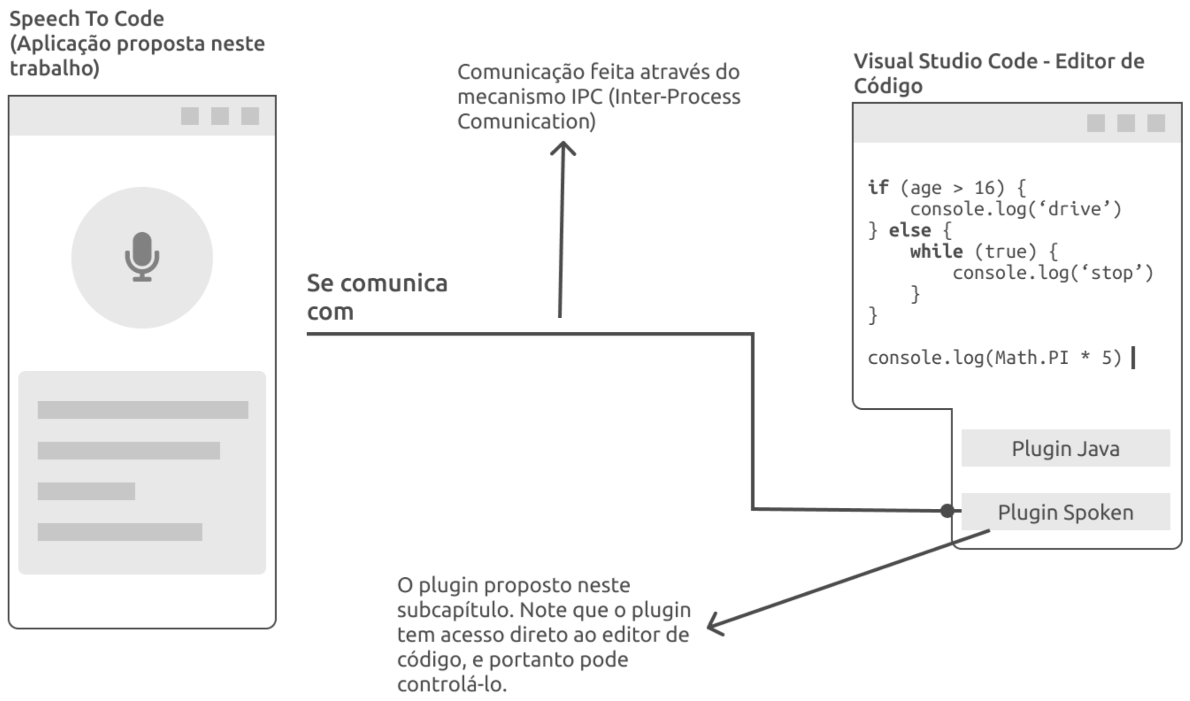
Figure 16. Diagram showing how the PyAutoGUI tool acts as an intermediary in communication between programs. Source: The author

2.3.2. EXTENSIONS

Plugins or extensions are an important feature of modern code editors that allow developers to create modules (plugins) capable of extending or adding new functionality to the code editor. Plugins run in the context of the code editor and therefore have access to internal APIs, which allow you to programmatically control it. Plugins can be used to add functionality such as: syntax highlighting for a programming language, code suggestion and completion, automatic closing of curly braces, parentheses and square brackets and so on.

Among the code editors and integrated development environments that support plugins and extensions are: *Visual Studio Code, Sublime Text, Intellij, Eclipse* and *NetBeans.*

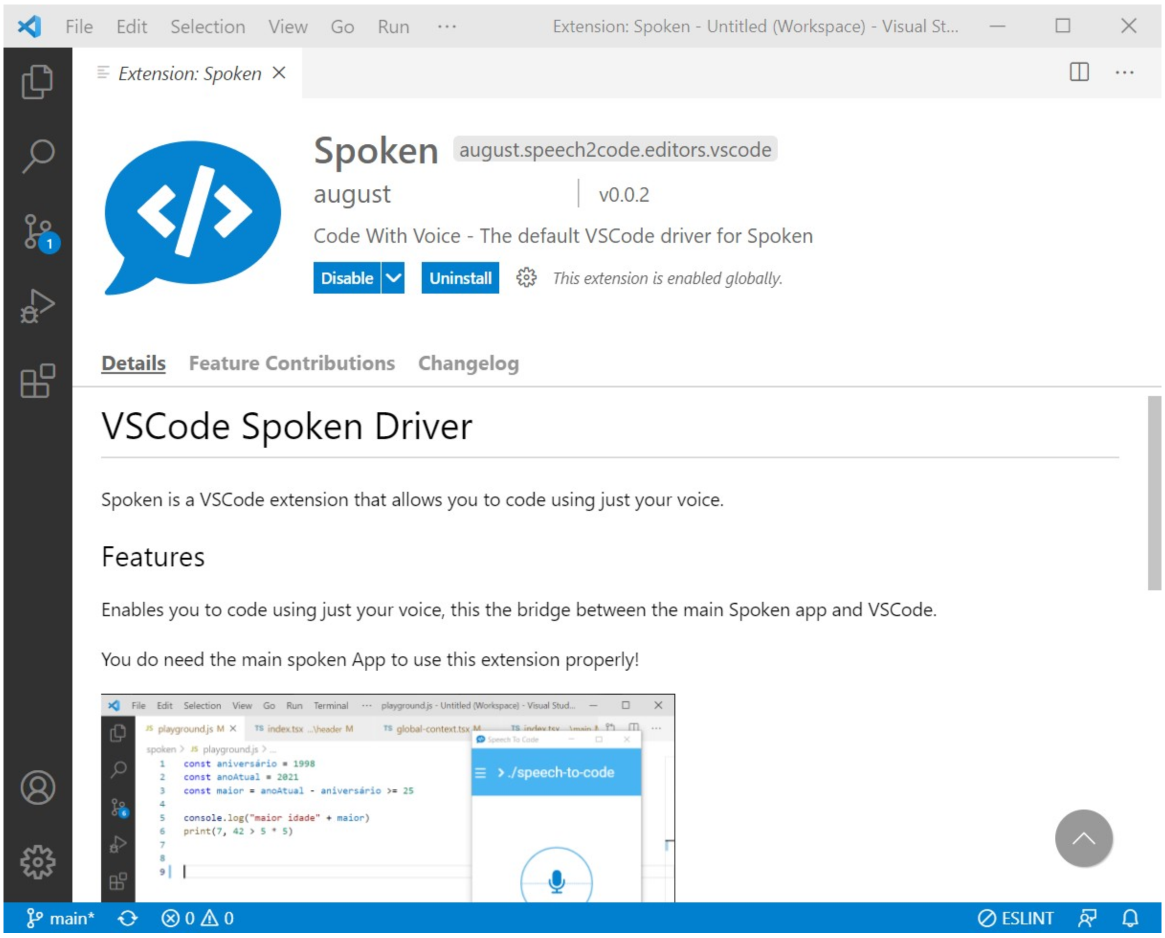
In this project, an extension/plugin for the code editor will be developed. *Visual Studio Code* able to control it based on commands received from an external application.


Speech To Code
(Aplicação proposta neste trabalho)
Comunicação feita através do mecanismo IPC (Inter-Process Comunication)
Visual Studio Code - Editor de Código
Se comunica com
Plugin Java
Plugin Spoken
O plugin proposto neste subcapítulo. Note que o plugin tem acesso direto ao editor de código, e portanto pode controlá-lo.

```
if (age > 16) {
    console.log('drive')
} else {
    while (true) {
        console.log('stop')
    }
}

console.log(Math.PI * 5)
```

Figure 17. Diagram showing how a Visual Studio Code plugin can communicate with a external application. Source: The author

2.3.2.1 Spoken Extension for Visual Studio Code



Spoken august.speech2code.editors.vscode

august | v0.0.2

Code With Voice - The default VSCode driver for Spoken

This extension is enabled globally.

VSCode Spoken Driver

Spoken is a VSCode extension that allows you to code using just your voice.

Features

Enables you to code using just your voice, this the bridge between the main Spoken app and VSCode.

You do need the main spoken App to use this extension properly!

```
const aniversário = 1998
const anoAtual = 2021
const maior = anoAtual - aniversário >= 25

console.log("maior idade" + maior)
print(7, 42 > 5 * 5)
```

Figure 18. Spoken: The extension for VSCode proposed and developed in this project. Source:*print screen* of the Spoken extension

Spoken is the name of the proposed extension developed in this project for the Visual Studio Code (VSCode) code editor. The purpose of this extension is to act as an intermediary between an external program that wants to perform operations on Visual Studio Code and Visual Studio Code itself. As shown in Figure 17, this extension is capable of receiving requests from an external program for manipulating the code editor and handling it as specified in the request.

As it is an extension, it runs in the same context as VSCode and has access to APIs that allow it to be controlled. Using these APIs, which only extensions have access to, it is possible to control (without user interaction) the VSCode to perform tasks such as: changing lines, writing text on any line, indenting the code, executing the current file, among many others .

This extension obeys a contract that lists what interactions it must be able to perform with the code editor. These interactions can be thought of as functions and some of them include:

Name	**Description**
write(String text)	Write something in the code editor
goToLine(int line)	Move pointer to specified line
select(int from, int to)	Selects text at specified coordinates.

Table 1. Some of the interactions that the extension is required to be able to perform with the editor of code.

External programs that want to interact with the code editor must also know this contract, they must be able to know what functions the extension, in this case Spoken, can perform in the code editor.

That said, the action associated with the command "execute ball function on line 42" can be interpreted as calling the function *goToLine(42),* which moves the editor pointer to line number 42, and *write("ball()"),* which writes the text “ball()” on the current line.

Finally, this extension is able to listen and execute requests coming from other programs through a file-based interprocess communication mechanism. Inter-process communication (IPC) is the mechanism that an operating system provides for processes to share data with each other. Applications that use IPC follow the client/server model, where the client requests data and the server responds to these requests (Microsoft Docs, 2018). In this project the extension will act as a server and all external programs in order to handle the VSCode as clients.

So if any application A wants to interact with the VSCode to, for example, remove a line, all it has to do is send a request to the Spoken extension and the line will be removed. This request must contain the name of the function to be performed and its arguments, as seen in table 1.

2.4. JAVASCRIPT ECOSYSTEM

The JavaScript programming language was chosen for the implementation of this project due to its ease of use, vast ecosystem and comprehensive documentation. Thanks to the vast ecosystem, it was possible to use the same programming language on all fronts of the project: either in the mechanism capable of recognizing and analyzing voice commands, in the mechanism capable of controlling code editors or in the application's user interface.

Below are listed the JavaScript frameworks used in creating the application's user interface.

2.4.1. Electron Framework

O *electron* it is a *framework* multiplatform developed by *GitHub.* Its goal is to allow the construction of desktop applications using technologies used in web development, such as *HTML, CSS* and *JavaScript*
(Noleto, 2020).

For this, he uses the *Chromium,* which is the open source version of the browser *Chrome.* The idea is that the developed application runs on it as if it were a web application. Also, the*electron* also uses the *Node.js,*
component that provides additional features to access internal operating system functions, such as allowing access to directories and files (Noleto).

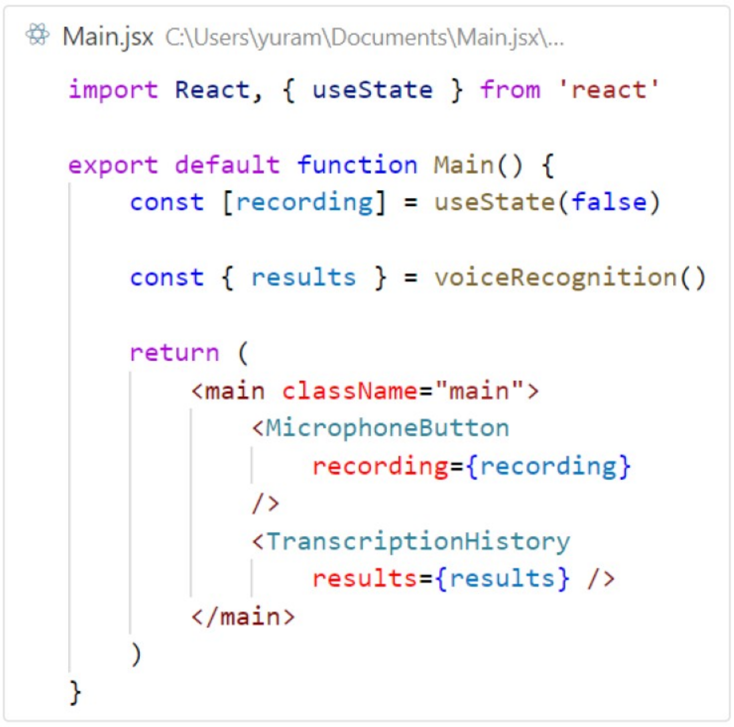
In a simplified way the *electron* is a framework that allows applications designed for the web, to be accessed using a browser, to work natively on desktops. Examples of applications that make use of the framework *electron* include the Whatsapp messaging application client and the code editor itself *Visual Studio Code.*

In the context of this project, one of the main advantages of using the framework *electron* is simplified access to the computer's microphone. On the web, access to the user's microphone and camera is standardized through the*MediaDevices* API. With *electron* it is possible to use this same API in desktop applications.

2.4.2. React Framework

ReactJS is a JavaScript library for creating user interfaces developed by Facebook in 2011 that simplifies the development process by introducing the concept of reusable reactive components. Using ReactJS it is possible to think of each element of a web page as its own component with an internal state that can be reused in other parts of the code. Thanks to its performance and simplicity the *ReactJS* was considered the second most popular framework in the year 2020, behind only the *JQuery,* according to Stackoverflow research.

That said, in this project the library will be used *ReactJS* for creating all UI components.



```
Main.jsx C:\Users\yuram\Documents\Main.jsx\...
import React, { useState } from 'react'

export default function Main() {
    const [recording] = useState(false)

    const { results } = voiceRecognition()

    return (
        <main className="main">
            <MicrophoneButton
                recording={recording}
            />
            <TranscriptionHistory
                results={results} />
        </main>
    )
}
```

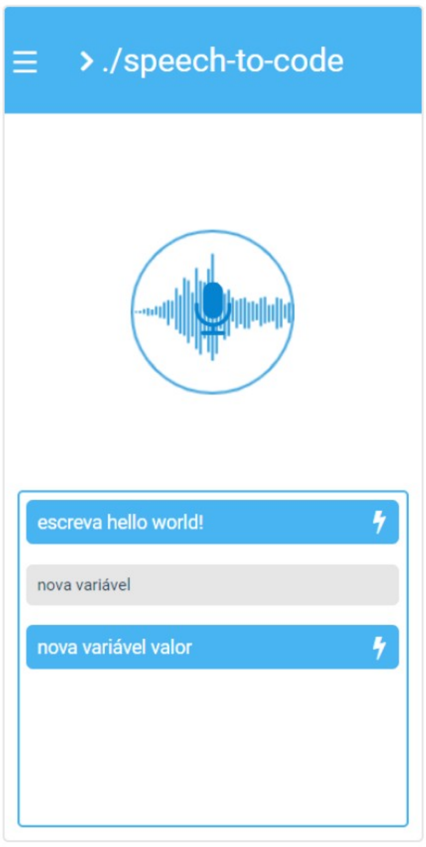


Figure 19. On the left the code in ReactJS used to create the application's splash screen, on the right the result. Source: The author

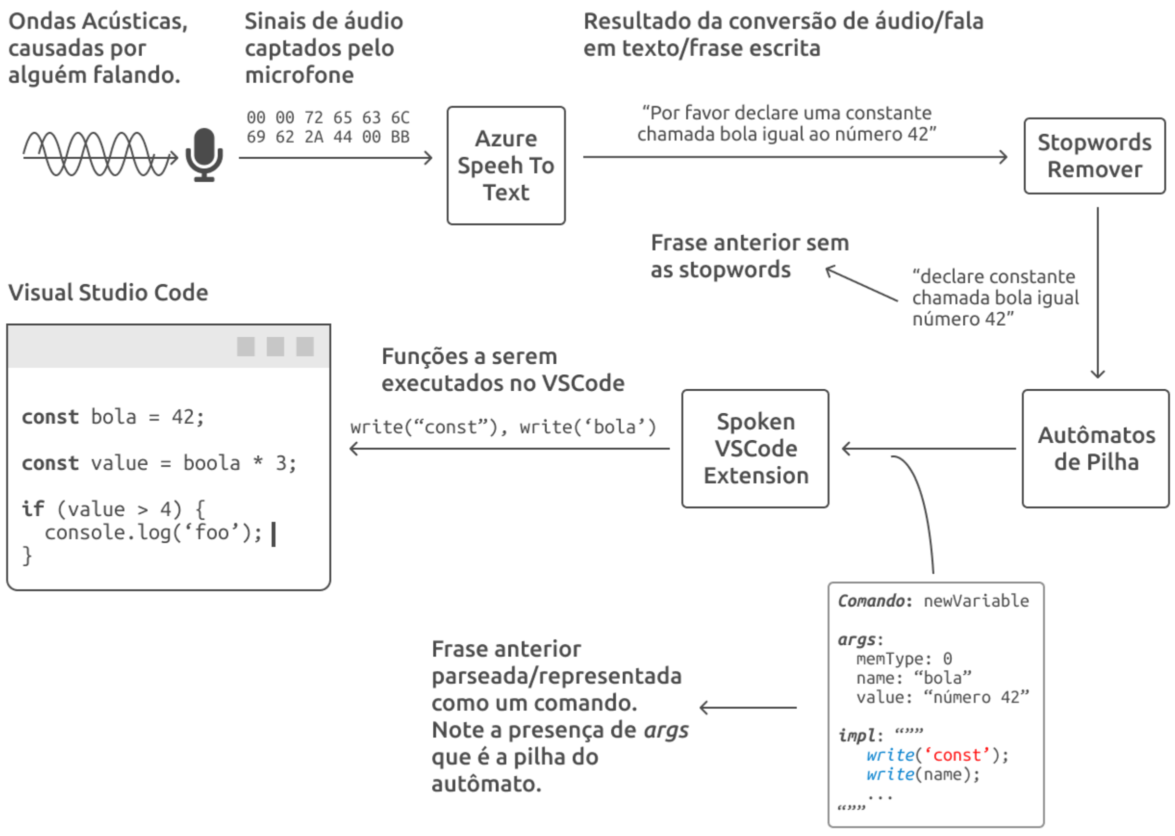
2.4.3. Express Framework

Express.js, or Express, was the framework used to develop the application's backend. O*Express* is one of the most popular frameworks in server development thanks to its simplicity of use and its vast ecosystem of modules that add punctual functionality to the framework.

O *Express* it is quite minimalist, however, it is possible to create specific middleware packages with the objective of solving specific problems that arise in the development of an application. There are libraries to work with cookies, sessions, user logins, URL parameters, data in POST requests, security header and etc (MDN Web Docs, 2020).

In this project the application backend, implemented with the framework *express,* will be responsible for serving the static files generated by the *ReactJS* and implement a token management service for the use of the service. *Azure Speech To Text.*

3. DEVELOPMENT OF THE SPEECH2CODE APPLICATION


Ondas Acústicas, causadas por alguém falando.
Sinais de áudio captados pelo microfone
00 00 72 65 63 6C 69 62 2A 44 00 BB
Azure Speeh To Text
Resultado da conversão de áudio/fala em texto/frase escrita
"Por favor declare uma constante chamada bola igual ao número 42"
Stopwords Remover
Frase anterior sem as stopwords
"declare constante chamada bola igual número 42"
Visual Studio Code
Funções a serem executados no VSCode
write("const"), write('bola')
Spoken VSCode Extension
Autômatos de Pilha
Frase anterior parseada/representada como um comando. Note a presença de *args* que é a pilha do autômato.

```
const bola = 42;

const value = boola * 3;

if (value > 4) {
  console.log('foo');
}
```

```
Comando: newVariable

args:
  memType: 0
  name: "bola"
  value: "número 42"

impl: """
    write('const');
    write(name);
    ...
"""
```

Figure 20. Application blueprint showing how all components interact with each other. Source: The author

Next, it will be shown how the concepts and technologies described in the previous chapter interact with each other to achieve the proposed objectives of this work.

The development process will be divided into 3 phases: definition, recognition and analysis of voice commands, *Spoken VSCode Extension* and creating the application's user interface.

3.1. VOICE COMMANDS

As seen earlier, voice commands are phrases that the user can say aloud that result in an action in the code editor. Next, the definition and how to create a voice command and the list of commands implemented in this project will be presented.

3.1.1. DEFINITION

Any command C can be seen as a pair (A, P) where:

- A is any action that must be performed when this command is triggered.
- P is a set of automatons capable of recognizing phrases associated with this command.

Therefore, if any phrase x is recognized by one of P's automata, action A must be triggered.

Physically a command can be interpreted as a folder containing two types of files:

- A single JavaScript file named “*impl.js*” containing instructions for carrying out the command.
- One or more files containing the textual representation of the stack automata used to recognize phrases associated with this command. These automata are represented using the graph representation language*DOT.*

Again, the instructions in the “*impl.js*” will only be executed for a sentence if it is recognized by one of the automatons that are represented by the files *DOT.*

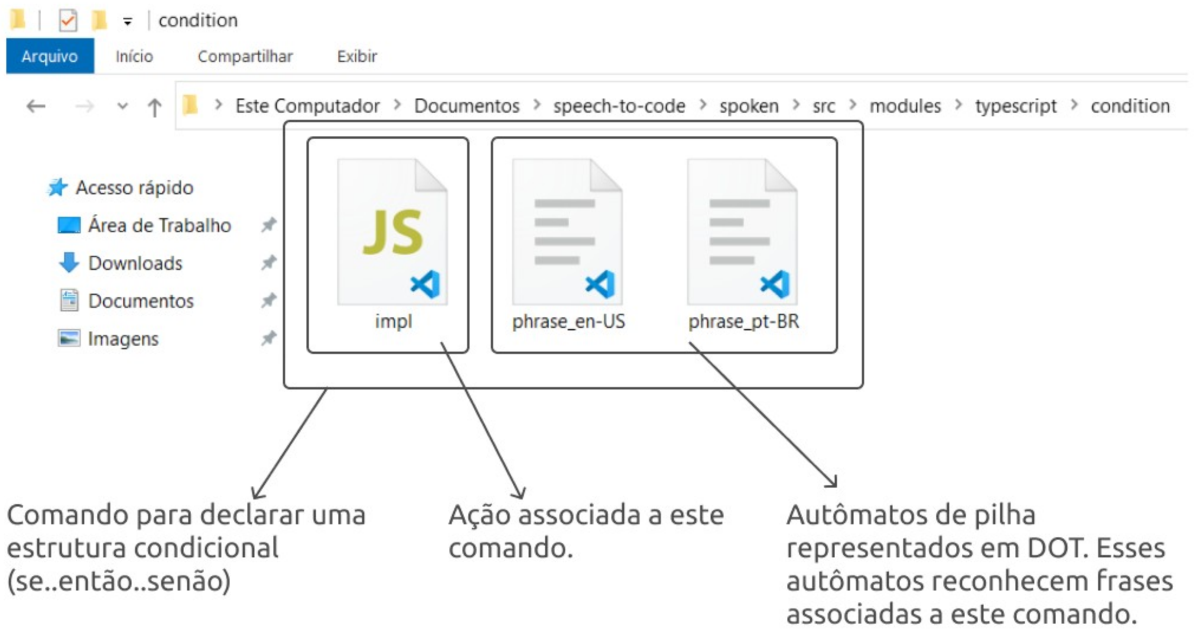
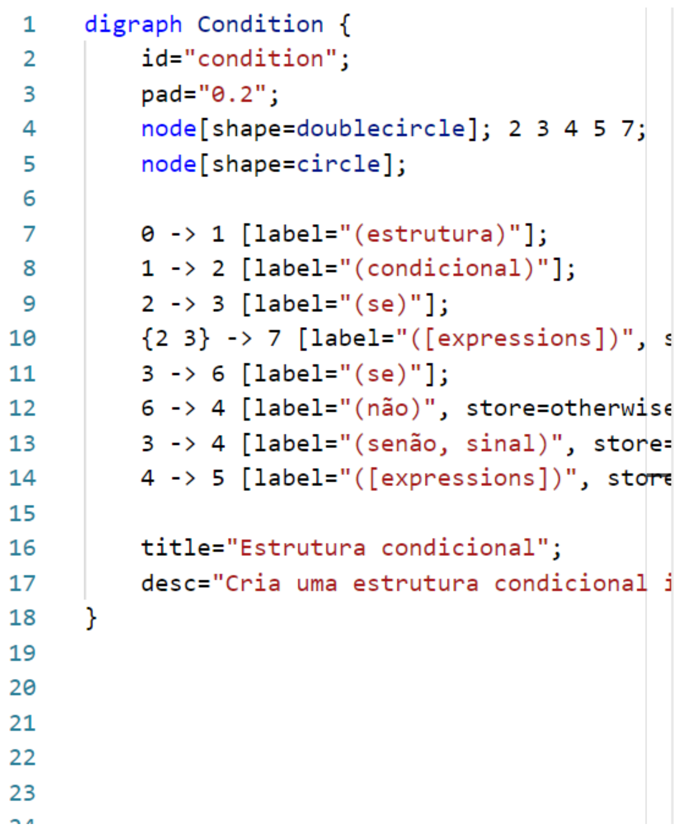

condition
Arquivo
Início
Compartilhar
Exibir
Este Computador > Documentos > speech-to-code > spoken > src > modules > typescript > condition
Acesso rápido
Área de Trabalho
Downloads
Documentos
Imagens
JS
impl
phrase_en-US
phrase_pt-BR
Comando para declarar uma estrutura condicional (se..então..senão)
Ação associada a este comando.
Autômatos de pilha representados em DOT. Esses autômatos reconhecem frases associadas a este comando.

Figure 21. Folder showing the structure of a command. Source: The author

In figure 20, which shows the definition of the command “conditional structure”, it is possible to observe the two types of files mentioned above. A call *"impl.js"* which represents the action to be performed by this command and two other calls *"phrase_en-US.dot"* and *"phrase_pt-BR.dot"* which represent the automatons responsible for recognizing phrases associated with this command.

The files "*impl.js*" and "*phrase_ptBR.dot*" (DOT file).

3.1.1.1. DOT files



```
digraph Condition {
    id="condition";
    pad="0.2";
    node[shape=doublecircle]; 2 3 4 5 7;
    node[shape=circle];

    0 -> 1 [label="(estrutura)"];
    1 -> 2 [label="(condicional)"];
    2 -> 3 [label="(se)"];
    {2 3} -> 7 [label="([expressions])", s
    3 -> 6 [label="(se)"];
    6 -> 4 [label="(não)", store=otherwise
    3 -> 4 [label="(senão, sinal)", store=
    4 -> 5 [label="([expressions])", store

    title="Estrutura condicional";
    desc="Cria uma estrutura condicional i
}
```

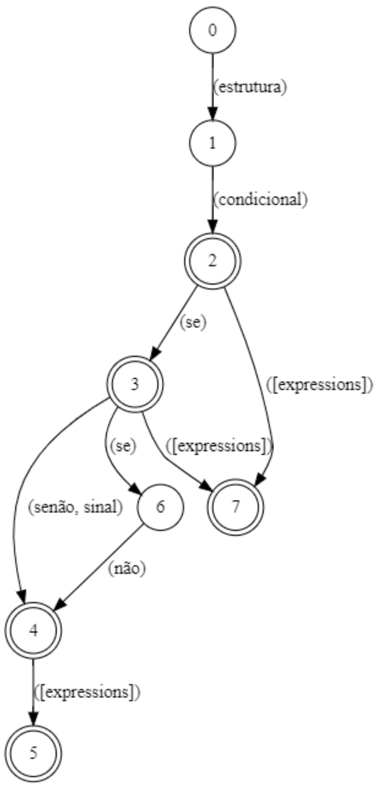


Figure 22. Contents of the “phrase_pt-BR.dot” file. On the left its textual representation and the visual right. Source: The author

The files “*phrase_en-US.dot*" and "*phrase_pt-BR.dot*” are the textual representation of a stack automaton capable of recognizing phrases related to the “conditional structure” command. These files were created using the graph representation language called DOT.

The automaton represented by the file "*phrase_en-US.dot*” only recognizes sentences in English while the “*phrase_pt-BR.dot*” only recognizes sentences in Portuguese. If there was a need to support a third language, say French, it could easily be added with the addition of a third automaton to recognize French phrases called “*phrase_fr-FR.dot*”. The idea is that it doesn't matter if the entry phrase is "*if five is greater than nine then*” (Portuguese) or “*if five is greater than nine then* ” (English) or if “*si cinq est supérieur à neuf alors*” (French) the action to be performed is always the same: the one associated with the “conditional structure” command. Currently the application supports English and Portuguese languages.

The stack automaton in figure 22 is able to recognize 8 different sentences related to creating a conditional structure, some of which include:

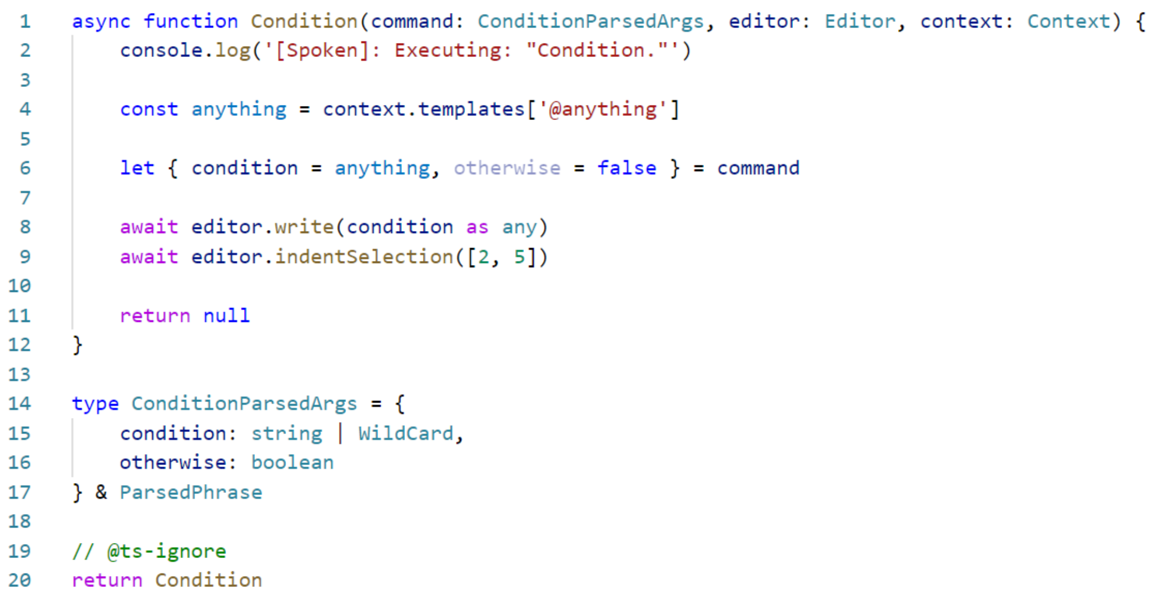
Phrase	**Battery**
conditional structure	(O)
conditional structure if but	(if not)
conditional structure [*expression]**	([expression]*)
conditional structure if otherwise [*expression]**	(otherwise, [expression])*

Table 2. Some of the phrases recognized by the automaton in figure 21.

(*) Any phrase recognized by an automaton called *expression.*

Any of these phrases is capable of triggering the action associated with the “conditional structure” command. As we saw earlier, the implementation of this action is found in the file "*impl.js*".

3.1.1.2. Implementation File



```
async function Condition(command: ConditionParsedArgs, editor: Editor, context: Context) {
    console.log('[Spoken]: Executing: "Condition."')

    const anything = context.templates['@anything']

    let { condition = anything, otherwise = false } = command

    await editor.write(condition as any)
    await editor.indentSelection([2, 5])

    return null
}

type ConditionParsedArgs = {
    condition: string | WildCard,
    otherwise: boolean
} & ParsedPhrase

// @ts-ignore
return Condition
```

Figure 23. Contents of the “impl.js” file. Source: The author

A JavaScript file containing just a function representing the action associated with the voice command, this function is executed whenever one of the associated automata recognizes a phrase as belonging to this command.

This function aims to perform what was asked in the voice command, for that it has access to the stack of the automaton that called it and the ability to interact with the *VSCode* using the protocol described in table 1. With the ability to interact with the *VSCode* this function can do things like write to the screen and switch lines and with access to the automaton's stack it knows *what to write on the screen* and *which line to go.*

Returning to the automaton in figure 21 and with the definition of action presented in this subchapter, we can complete table 2 as follows:

Phrase	**Battery**	**Action**

conditional structure	(O)	Write on the screen:
		
		if (anything) {
		}

conditional structure if not	(if not)	Write on the screen:
		
		if (anything) {
		} else {
		}

conditional structure [*expression]**	([expression]*)	Write on the screen:
		
		if ([expression]) {
		}

conditional structure if not *[expression]**	(otherwise, [expression])*	Write on the screen:
		
		if ([expression]) {
		} else {
		}

Table 3. Some of the phrases recognized by the automaton in figure 21 and its actions in the VSCode.

(*) Any phrase recognized by an automaton called *expression.*

With this we can conclude that the action taken by a command depends on which phrase was used to trigger that command, in other words, given a phrase and an action, the action performed will depend on the contents of the automaton's stack that recognized that phrase.

3.1.2. List of Implemented Commands

In the previous subchapter it was demonstrated that creating a command is actually a process of creating an automaton to recognize phrases and an action that must be performed when a phrase is recognized. Listed below are which commands were implemented using this method, along with a brief description and example sentences in Portuguese and English that can be used to activate each command.

It is worth noting that all actions are applied to the code editor. *Visual Studio Code* and the output language is JavaScript. So when the description of a command says "declare a variable" it should be interpreted as "declare a variable in the *Visual Studio Code* using programming language syntax
JavaScript”.

Name	Examples of sentences (pt-BR en-US)	Description
variableAssign	“new constant b equal to number 7”, “new called constant graph”, "constant b equals string hello string"	Declare a variable with name and value specified.
variableRef	"constant ball reference", "constant graph"	References an already declared variable.
number	"please number 32", "number 4646", "number 99"	Write the specified integer.
string	"string your age is string", "string what is that string", "text only a text test"	write the string specified. Everything between the word string or text is considered as a string.
fncall	“run the sum function with arguments number 10 and string hello string”, "call function list on variable people"	Call a function with name, arguments and *caller* specified.
mathexp	“expression number 4 plus variable boi expression minus execute the list function”,	Creates an math with the

	"expression number 4 minus string hello string"	specified values. Supports the operations: addition, subtraction, division and multiplication.
logicExp	“expression execute function * calculate imc * greater than the number 10”, “expression variable b and variable c”	Create a logical expression with the values specified. Supports: and, or, >, <, >=, <=.
condition	"conditional structure if expression number 5 greater than variable ball", "conditional statement if else expression call function value or variable ball"	Create a structure conditional if...then with the specified condition.
goToLine	“go to line 4 please”, “line 10”, "line number 4",	Changes the current line to the specified line.
newLine	"new line", "new line", "new line, please"	Add one more line.
select	"select from line 3 to line 10", "select word value", "select from letter e to letter j"	Selects the range between specified lines, letters and words.
run	“run this file please”, “please run this file”	Run the code from the current file.
newFn	“new ball function with 3 arguments”, “new function sum”, "new hi function returning number 4"	Creates a function with name, return value and number of arguments.
repetition	"repeat structure", "repeat structure from number 4 to number 43" "repeat structure for every item in the list"	Creates a repeat loop with the arguments provided.
write	"please write how old you are", "please write it down who are you"	Write anything in the code editor. All that after the

	"can you please print hello doctor"	word "write" will be written.

Table 4. Commands implemented during this project.

The complete list of these commands along with a more in-depth description, their implementations and definition of the stack automata can be found at: *https://github.com/pedrooaugusto/speech-to-code/tree/main/spoken/src/modules/typescript#typescript-voice-commands*

3.1.3. Transforming Voice Commands into Text

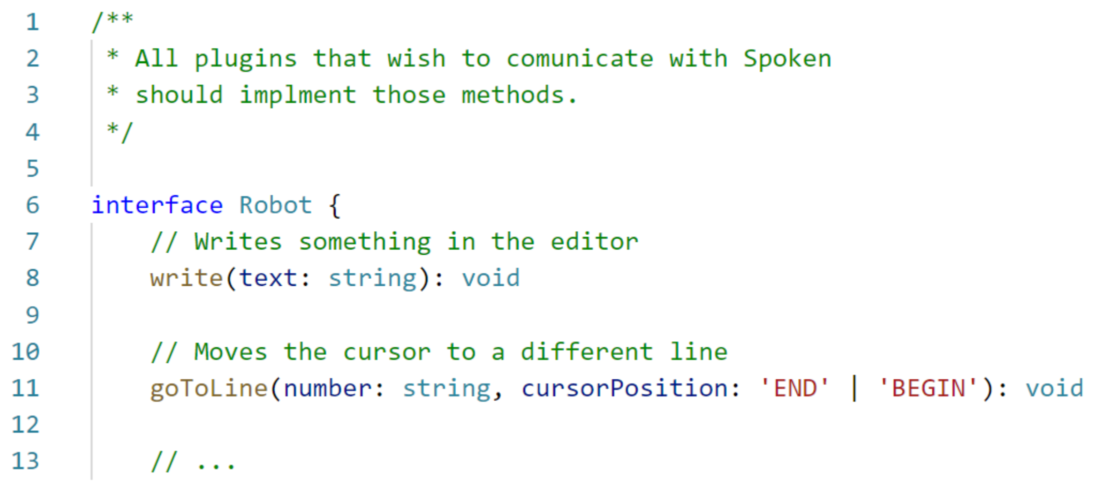
Until now, voice commands have been treated as sentences in textual format, in this subchapter we will see how to transform sentences spoken aloud to text, so that they can then be recognized by the aforementioned automaton engine.

To transform sentences spoken by the user into text will be used a Microsoft product called *Azure Text To Speech,* this product gives you access to Microsoft's speech-to-text conversion service, the same one used in your voice assistant *Cortana (*Azure, 2020). Because it is a*web service* it can be used on any device with internet access and works on client/server model over *websockets.*

In this project a functionality of the *Azure Speech To Text* capable of real-time voice recognition on the audio picked up by the device's microphone. Every second the client sends a packet containing 1 second of audio captured by the microphone to the voice recognition service which, based on all audio packets already received, tries to identify what was said, and then sends back to the client the your best*guess* of what was said in textual form.

In the context of this project, after receiving this hunch of what was said in textual form, all the automata mechanism already described will be used to test whether what was said is also a valid command.

3.2. Spoken VSCode Extension

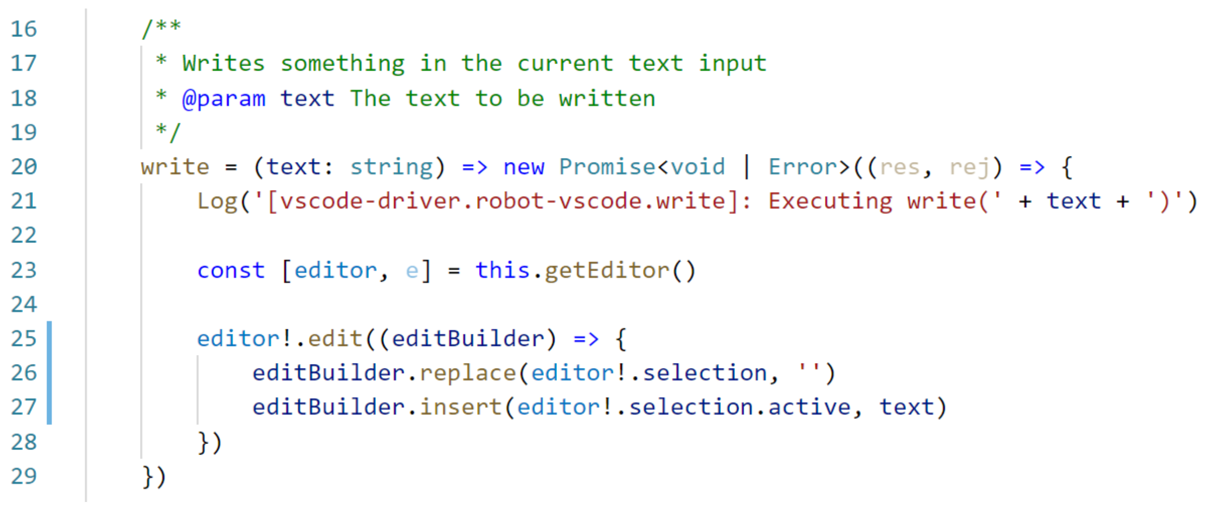


```
/**
 * All plugins that wish to comunicate with Spoken
 * should implment those methods.
 */

interface Robot {
    // Writes something in the editor
    write(text: string): void

    // Moves the cursor to a different line
    goToLine(number: string, cursorPosition: 'END' | 'BEGIN'): void

    // ...
```

Figure 24. Example of some methods that the extension must implement by contract.

Source: The author

The extension development process *Spoken* it consists of implementing all the methods it must support as set out in the contract. By following this contract it is guaranteed that any command in table 3 can be successfully executed by this extension. An example of this contract can be seen in Figure 23 which says that the extension*Spoken* must be able to write something in the code editor. The next step is to do just that using the API that VSCode exposes for your extensions as shown in the figure below.



```
/**
 * Writes something in the current text input
 * @param text The text to be written
 */
write = (text: string) => new Promise<void | Error>((res, rej) => {
    Log('[vscode-driver.robot-vscode.write]: Executing write(' + text + ')')

    const [editor, e] = this.getEditor()

    editor!.edit((editBuilder) => {
        editBuilder.replace(editor!.selection, '')
        editBuilder.insert(editor!.selection.active, text)
    })
})
```

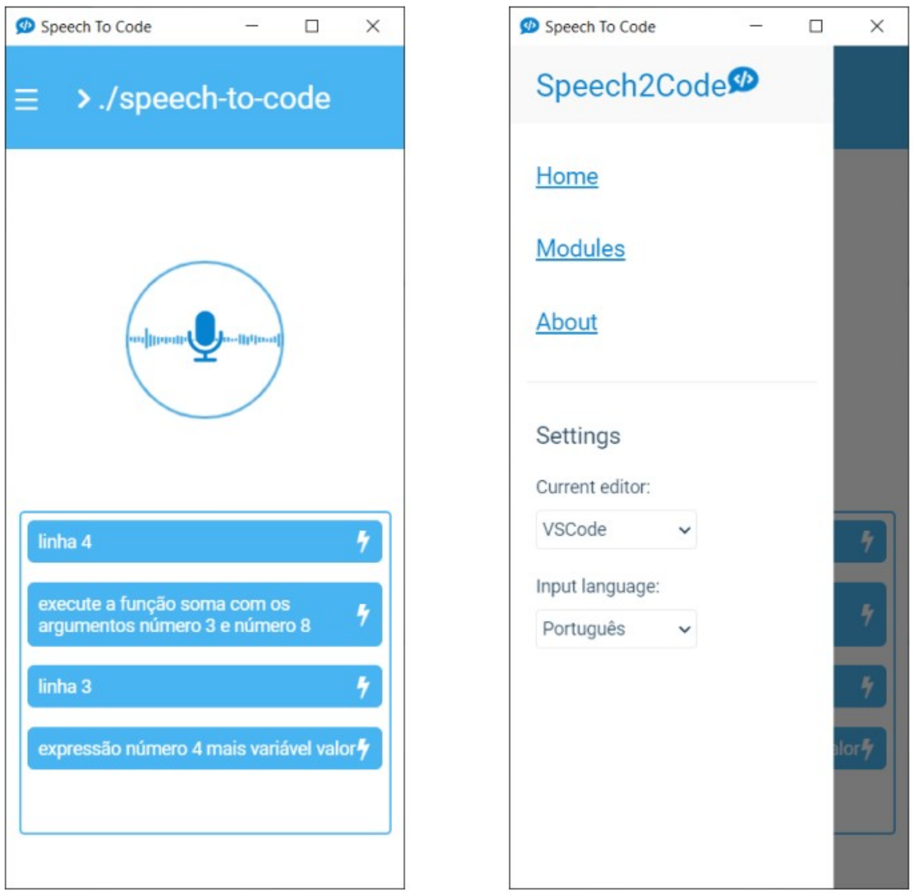
Figure 25. Implementation of write method using VSCode APIs. Source: The author

Another important feature of this extension is the ability to accept requests from other applications to handle the VSCode. This is achieved using the inter-process communication method. When starting the extension opens a communication channel called "*speechtocodechannel*" that any application can connect and “chat”. It is through this channel that external applications send the commands from table 1 to be executed in the context of VSCode.

3.3. USER INTERFACE

All screens in the application were created with simplicity and usability in mind. Screens were first designed in the design tool*figma* and then implemented using the aforementioned UI build library *ReactJS.* Next, the screens will be shown and the functionalities of each one will be discussed.

3.3.1. Home screen



linha 4

execute a função soma com os argumentos número 3 e número 8

linha 3

expressão número 4 mais variável valor

Current editor: VSCode

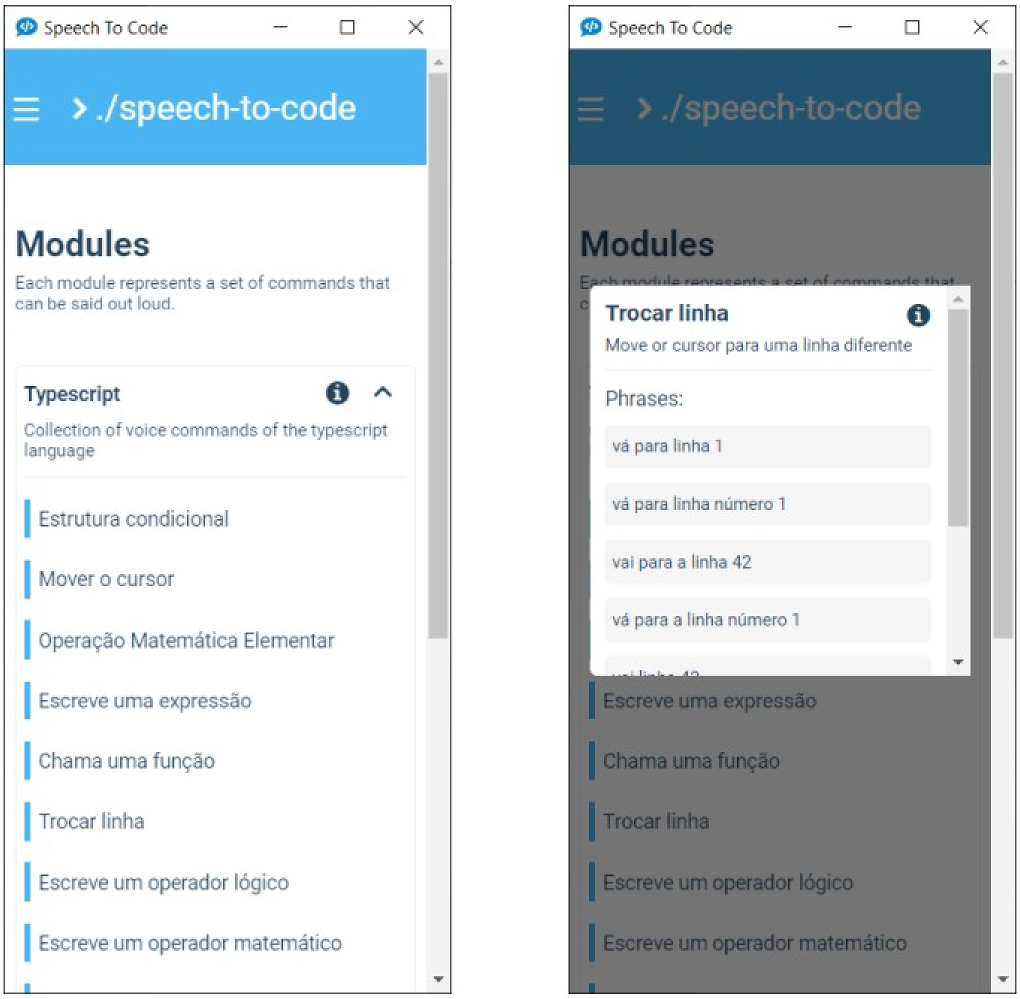
Input language: Português

Figure 26. Application home screen. Left in its natural state and right with expanded side menu. Source: application print screen developed.

On the left in the figure above, you can see the two main components that make up the initial screen: a circle with a microphone design and below a list with a series of sentences. The circle is actually a button that when clicked starts the voice recognition process which only ends when the button is clicked again, to denote that everything being said is being recorded and analyzed the button will turn red. Everything that is said while the speech recognition process is active will be turned into text and will appear in the phrase list, the phrases that are also recognized as commands will appear in blue color, otherwise gray. In the figure we can see that all the said sentences are also valid commands, as they are in blue color.

On the right of figure 26 we can see the main menu of the application which has 3 basic actions: change the application language between portuguese and english, change the code editor that will be controlled and go to the modules page.

3.3.2. Modules Screen


Speech To Code

>./speech-to-code

Modules

Each module represents a set of commands that can be said out loud.

Typescript

Collection of voice commands of the typescript language

Estrutura condicional

Mover o cursor

Operação Matemática Elementar

Escreve uma expressão

Chama uma função

Trocar linha

Escreve um operador lógico

Escreve um operador matemático

Speech To Code

>./speech-to-code

Modules

Trocar linha

Move or cursor para uma linha diferente

Phrases:

vá para linha 1

vá para linha número 1

vai para a linha 42

vá para a linha número 1

Escreve uma expressão

Chama uma função

Trocar linha

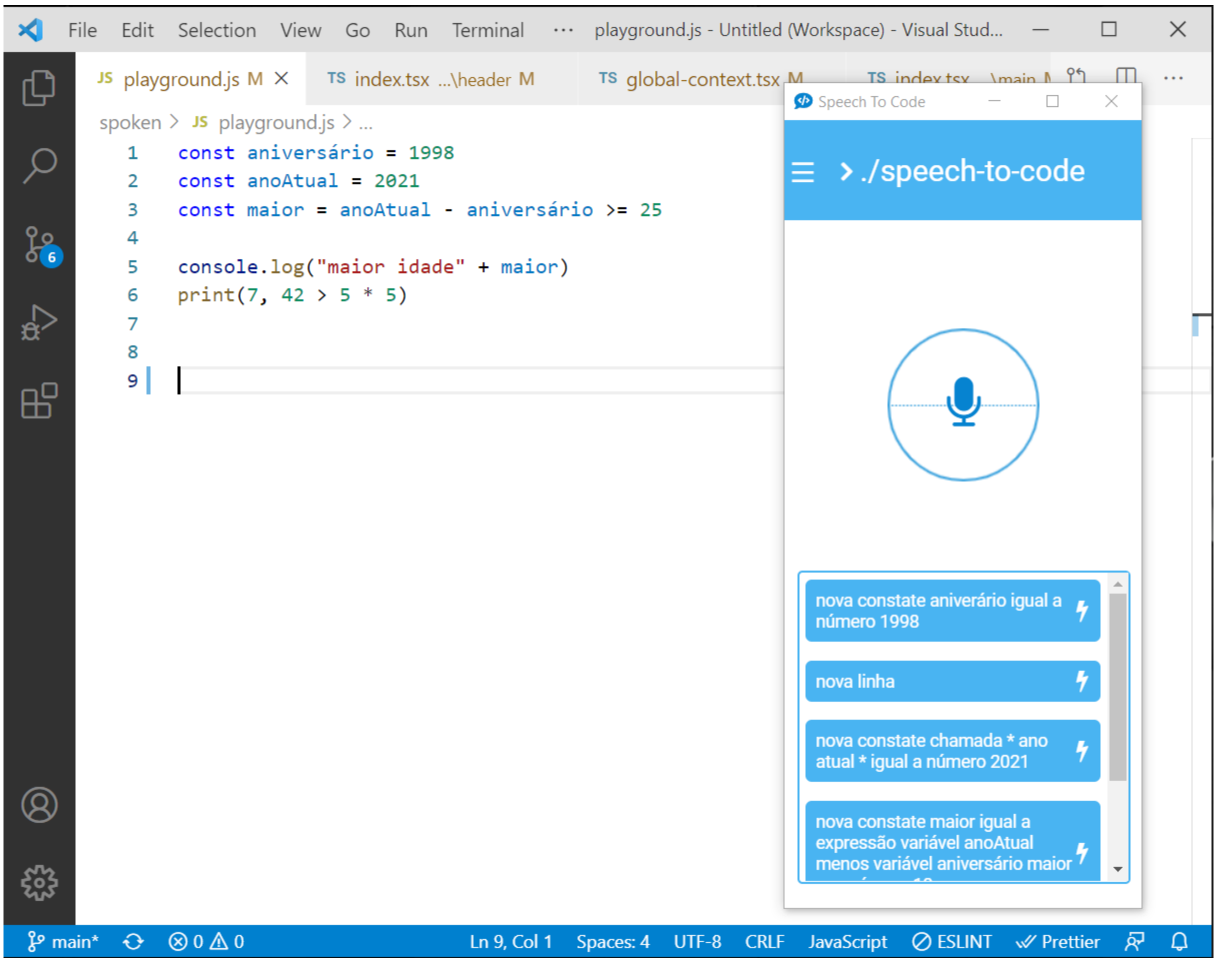
Escreve um operador lógico

Escreve um operador matemático

Figure 27. Modules screen. Lists all application commands. On the left a list with all available commands, on the right command details to switch lines. Source:*print screen* of the developed application

The modules screen lists and shows details of all commands available in the application and is also sensitive to the current application language, that is, if the application language is in English only the commands available in English will be listed. Through this screen it is also possible to follow a link to more information about each command.

4. RESULTS



```
const aniversário = 1998
const anoAtual = 2021
const maior = anoAtual - aniversário >= 25

console.log("maior idade" + maior)
print(7, 42 > 5 * 5)
```

nova constate aniverário igual a número 1998

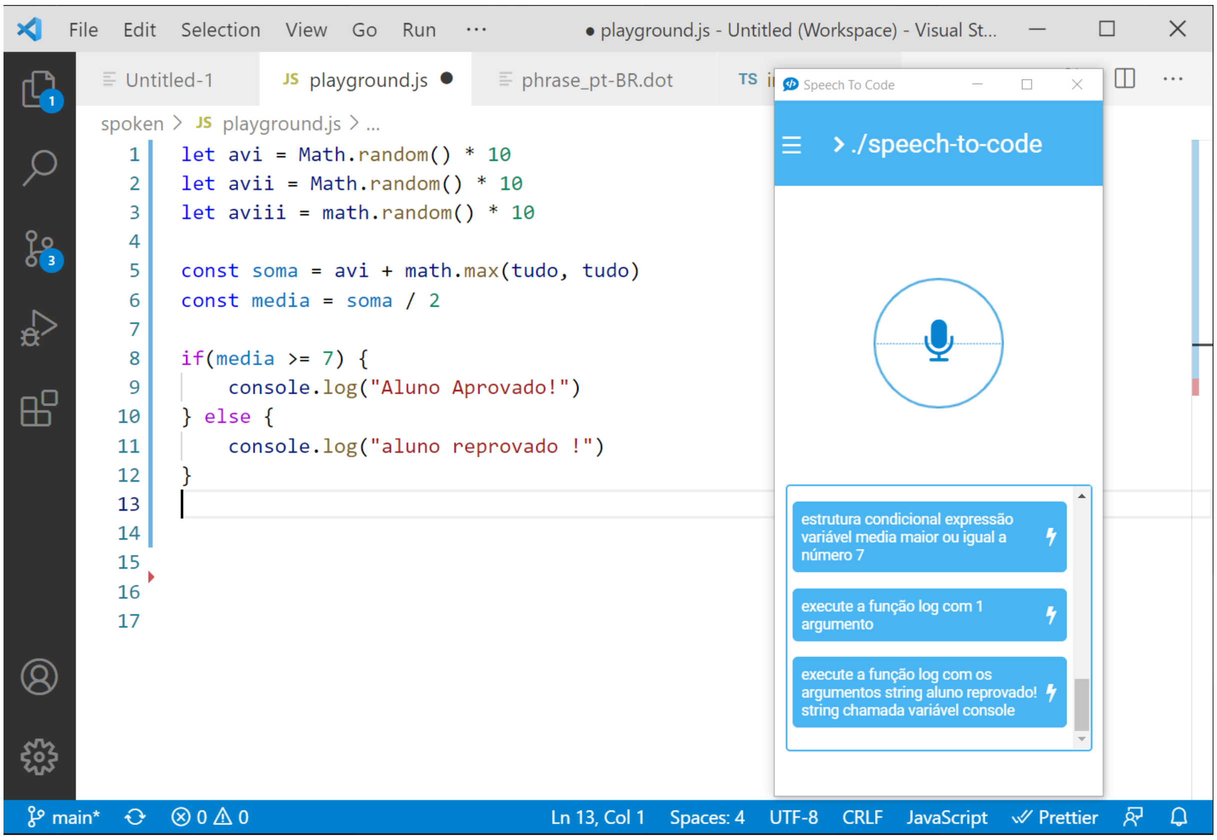
nova linha

nova constate chamada * ano atual * igual a número 2021

nova constate maior igual a expressão variável anoAtual menos variável aniversário maior

Figure 28. Example of the application in operation. On the left you can see the generated code and on the right the voice commands used. Source:*print screen* of the application developed interacting with the VSCode.

The initial purpose of this work was to create a tool capable of enabling programmers to program in JavaScript using voice. For this, we developed: a context-free language representing the available voice commands (table 3), an application capable of identifying voice commands belonging to this language (figure 26) and an extension capable of controlling the code editor *Visual Studio Code* to execute such a command (figure 18). The result of all these components working together can be seen in the figure above which shows an excerpt of code written entirely using speech. You can see in the lower right corner that at some point the phrase "*declare constant call birthday equal to number 1998*" was said aloud, and identified as a valid command. Therefore, the first line of the editor of code shown at bottom of image is "*const birthday = 1998*", exactly what was asked in the sentence.



```
let avi = Math.random() * 10
let avii = Math.random() * 10
let aviii = math.random() * 10

const soma = avi + math.max(tudo, tudo)
const media = soma / 2

if(media >= 7) {
    console.log("Aluno Aprovado!")
} else {
    console.log("aluno reprovado !")
}
```

estrutura condicional expressão variável media maior ou igual a número 7

execute a função log com 1 argumento

execute a função log com os argumentos string aluno reprovado! string chamada variável console

Figure 29. Application example at work. Source: application print screen

In the figure above, you can see a program to calculate the average of three grades and it shows on the screen "Approved", if the average is greater than or equal to 7, or "Failed", if the average is less than 7. When creating this program control structures, variable manipulation, function calls and arithmetic and logical expressions were used. The program was created using just voice commands.

As it is a general-purpose programming language and one of the most popular in the world today, JavaScript supports a considerable number of features, since *async/await* until *bitwise operations* and anonymous functions.

Due to this large number of features and the limited time to carry out this project, a basic set of features that the application must support was stipulated. That is, although the JavaScript language supports a considerable number of different features, using this application it will only be possible to use a limited set of these features. The number of supported features is directly related to table 3 and will grow at the same pace as this one.

The following will show the stipulated sets of basic functionality and whether or not they were implemented along with an example in the JavaScript language.

Set	**Description**	**Example**	**Implemented**

Manipulation of variable	declare and do reference to variables already declared.	const a=7;
		const b = 6;
		const c = a * b;
			YES.

Structures Conditional	Create structures conditionals.	if (3 > 4) {}
		else {}
		
		switch(a) {...}
			PARTIALLY.
			
			Supports: if...else
			Does not support: *shor-circuit* and *switch*

Roles	create and run a function with arguments.	Math.max(0, 1);
		hello();
		test("hi");
			YES.

Structures of repetition	Create structures of repetition.	for(...) {}
		while(...) {}
			YES.

Primitive Types	Manipulation of primitive types of language.	let a = "hello" let b = true let c = 42	PARTIALLY. Supports: numbers integers and strings. Does not support: null, undefined and booleans.
Logical operations and math.	Logical operations and math that manipulate 2 or more terms.	42 * 7 > 34 ~(a && b) 1 + 2 + 3 / 5	YES.

complex types	Allows the creation and manipulation of non-primitive types (classes/structs).	new Date(42)
		class Hello {...}
		{name: “pedro”}
			NO.

Table 5. Set of stipulated basic functionalities.

4.1. Requirements for running the application

The minimum requirements for the application to work fully are as follows:

- **Network:**
  Stable and fast internet connection;
- **Operating System:**
  Tested on Windows 10 only;
- **Memory:**
  At least 3GB of ram and 270MB of available disk space;
- **Python:**
  Tested with Python version 3.9.1;
- **Visual Studio Code:**
  Tested with version 1.55.2 of VSCode;

5. CONCLUSION

In this work, an assistive technology application capable of enabling programmers to program in JavaScript without the use of their hands, using only voice commands was proposed and developed. The purpose of the application is to enable programmers who are unable or with great difficulty to use their hands, whether due to injury or disability, to continue exercising their profession. Its development was based on the concepts of automata theory, speech recognition and user interface automation. This same methodology can be used in the creation of other applications with a similar purpose, such as a tool that, in addition to JavaScript, also supports Python or is capable of manipulating the linux bash terminal.

In addition to increasing the number of JavaScript language functionality supported, effectively completing Tables 3 and 4, future work includes some improvements in voice command recognition and parsing:

- Train and create a custom speech recognition template in *azure* *speech to text* only with the phrases that represent commands. This will make the speech recognizer prefer these phrases. An example of this is the phrase used to write something on the screen: "*write something*". By saying "*write test*" out loud is often recognized as "*right test*", because the words "*write*" and "*right*" have similar pronunciation. This can be resolved by creating a custom template or adding more context to the sentence.

- Evaluate replacing the mechanism for recognizing and analyzing commands, which is based on stack automata, with more robust machine learning-based solutions such as *Microsoft Azure LUIS* or the *Google Cloud Natural Language AI.*

BIBLIOGRAPHIC REFERENCES

PUPO, Deise Tallarico; MELO, Amanda Meincke; PEREZ FERRÉS, Sofia. **Accessibility: Discourse and Practice in the Daily Life of Libraries.** Sao Paulo-SP: UNICAMP, 2008.

BERSCH, Rita. What is Assistive Technology?.**Assistive technology.** Available at: <https://www.assistiva.com.br/tassistiva.html>. Accessed on: May 5th. 2021.

Assistive Technology Australia. What is Assistive Technology?.**assisted technology Australia.** Available in: <https://at-aust.org/home/assistive_technology/assistive_technology>. Access on: 05 May. 2021.

SOCIAL PROTECTION, Social Security. Statistical data - Social Security and INSS: **Federal government,** 2019. Available at: <https://www.gov.br/previdencia/ptbr/acesso-a-informacao/dados-abertos/previdencia-social-regime-geral-inss/dadosabertos-previdencia-social>. Accessed: May 15, 2021.

BEGEL, Andrew. Programming by voice: A domain-specific application of speech recognition. **AVIOS speech technology symposium–SpeechTek West. [**SI], 2005.

IBM CLOUD, IBM Cloud Education. Speech Recognition:**IBM Cloud Education,** 2020. Available at: <https://www.ibm.com/cloud/learn/speech-recognition>. Accessed: May 15, 2021.

RABINER, Lawrence; JUANG Biing-Hwang. **Fundamentals of speech Recognition.** USA: Prentice Hall, 1993.

SILVA, Anderson. **Voice Recognition For Isolated Words.** Advisor: Ren Tsang. 2009. 60. Monograph – Computer Engineering, Federal University of Pernambuco, Pernambuco. 2009. Available at: <https://www.cin.ufpe.br/~tg/2009-2/ags.pdf>. Accessed on: May 15, 2021.

Wilbur, Karl Sirotkin. **The automatic identification of stop words.** Journal of Information Science, 1992.

John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. **Introduction to automata theory, languages, and computation.** Addison-Wesley, 2001.

Eric Roberts. **Basics of Automata Theory.** CS Stanford EDU, 2009. Available at: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automatatheory/basics.html>. Accessed: May 15, 2021.

Fr. Gouda, Prabhakar. **Application of Finite Automates.** Academic Dictionaries and Encyclopedias, 2009

Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. **String similarity measures and joins with synonyms.** USA: Association for Computing Machinery, 2013.

Emden R. Gansner, Eleftherios Koutsofios, Stephen North. **Drawing graphs with dot.** Graphviz Org, 2015. Available at: <https://www.graphviz.org/pdf/dotguide.pdf>. Accessed: May 15, 2021.

UIPATH, UIPATH. **What is robotic process automation?.** UI PATH, 2015. Available at: <https://www.uipath.com/rpa/robotic-process-automation>. Accessed: May 15, 2021.

Al-Sweigart. **Welcome to PyAutoGUI's documentation!.** PyAutoGUI, 2017. Available at: <https://pyautogui.readthedocs.io/en/latest/index.html>. Accessed: May 15, 2021.

Microsoft Docs. **Interprocess Communications.** Microsoft Docs, 2018. Available at: <https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications?redirectedfrom=MSDN>. Accessed: May 15, 2021.

Cairo Noleto. **Electron: What is it and how to create applications using this framework!.** Trybe, 2020. Available at: <https://blog.betrybe.com/framework-deprogramacao/electron/>. Accessed: May 15, 2021.

Stackverflow. **Most Popular Technologies: Frameworks.** Stackoverflow, 2020. Available in: <https://insights.stackoverflow.com/survey/2020#most-popular-technologies>. Accessed: May 15, 2021

MDN Web Docs. **Express/Node introduction.** Developer Mozilla, 2020. Available at: <https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introduction>. Accessed: May 18, 2021.

Azure. **Speech to Text Conversion.** Microsoft Azure, 2020. Available at: <https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/# overview>. Accessed: May 19, 2021.

Maria Bruna. **Repetitive strain injury (RSI/DORT).** Drauziovarella. Available in: <https://drauziovarella.uol.com.br/doencas-e-sintomas/lesao-por-esforco-repetitive-read-dort/>. Accessed: May 20, 20221.