CENTRO UNIVERSITÁRIO CARIOCA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PEDRO AUGUSTO RIBEIRO DA SILVA

FERRAMENTA DE TECNOLOGIA ASSISTIVA BASEADA EM VOZ CAPAZ DE AJUDAR PROGRAMADORES A PROGRAMAR NA LINGUAGEM JAVASCRIPT

RIO DE JANEIRO
2021

PEDRO AUGUSTO RIBEIRO DA SILVA

FERRAMENTA DE TECNOLOGIA ASSISTIVA BASEADA EM VOZ CAPAZ DE AJUDAR PROGRAMADORES A PROGRAMAR NA LINGUAGEM JAVASCRIPT

Trabalho de conclusão de curso apresentado ao Centro Universitário Carioca (Unicarioca), como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientador(a): Prof.ª Daisy Cristine Albuquerque da Silva, M.Sc.

RIO DE JANEIRO
2021

PEDRO AUGUSTO RIBEIRO DA SILVA

FERRAMENTA DE TECNOLOGIA ASSISTIVA BASEADA EM VOZ CAPAZ DE AJUDAR PROGRAMADORES A PROGRAMAR NA LINGUAGEM JAVASCRIPT

Aprovado por:

_____

Profa. M.Sc. Daisy Cristine Albuquerque da Silva
Centro Universitário Carioca – UniCarioca
(Orientadora)

_____

Prof. M.Sc. André Luiz Avelino Sobral
Centro Universitário Carioca – UniCarioca
(Coordenador)

_____

Prof. Antonio Felipe Podgorski Bezerra
Centro Universitário Carioca – UniCarioca
(Convidado)

RIO DE JANEIRO
2021

AGRADECIMENTOS

À Deus, pois sem ele nada é possível.
Aos meus pais, por todo o suporte e sacrifícios em prol de meu benefício.
À minha orientadora, por toda ajuda e paciência durante o desenvolvimento deste trabalho.
À Unicarioca, seu corpo docente e funcionários.

RESUMO

Lesão por esforço repetitivo (LER) é uma síndrome que afeta milhares de profissionais ao redor do mundo todos os anos, dentre esses profissionais encontram-se os programadores, profissionais estes que tendem a fazer uso extensivo das mãos na realização de suas tarefas e por isso mais susceptíveis a LER. Programadores afetados por esta síndrome tem grande dificuldade em usar as mãos e consequentemente grande dificuldade em realizar sua atividade principal: programar, atividade historicamente baseada em texto que requer uso extensivo das mãos. Dito isto, chegamos ao tema central deste trabalho: a criação de uma ferramenta de tecnologia assistiva capaz de auxiliar programadores a programar na linguagem JavaScript fazendo uso da voz. A ferramenta tem como objetivo habilitar programadores a programar sem o uso das mãos, usando apenas comandos de voz. Neste trabalho será abordado como transformar áudio em texto utilizando softwares de reconhecimento de fala, como identificar e *parsear* frases pertencentes a uma dada gramática utilizando autômatos de pilha, como conectar-se e controlar editores de código utilizando ferramentas de automação de interface de usuário, o processo de criação de uma extensão para o editor de código *Visual Studio Code* e a criação de uma aplicação multiplataforma utilizando ReactJS e o framework Electron.

Palavras chaves: Reconhecimento de voz, autômato de pilha, tecnologia assistiva, automação de interface de usuário, conversão de fala em código.

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 their tasks and that is why they are most 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 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. This work will cover how to transform audio into text using speech recognition software, how to identify and parse phrases belonging to a given grammar using a stack automaton, how to connect and control code editors using user interface automation tools, the process of creating an extension for the Visual Studio Code editor and the creation of a multiplatform application using ReactJS and the Electron the framework.

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

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TA - Tecnologia Assistiva

LER - Lesão Por Esforço repetitivo

VSCode - Visual Studio Code

STT - Speech To Text

AST - Automatic Speech Recognition

API - Application Programming Interface

SDK - Kit de Desenvolvimento de Software

AFD - Autômato finito determinístico

LIFO - Last-In-First-Out

DOT - Linguagem para representação de grafos em formato textual

GUI - Graphical User Interface

IPC - Inter Process Communication

HTML - Hypertext Markup Language

CSS - Cascading Style Sheets

POST - Um dos tipos de uma requisição HTTP

SUMÁRIO

1. INTRODUÇÃO

Muito tem se falado sobre os chamados softwares de acessibilidade, softwares estes que visam facilitar o uso de computadores e equipamentos por pessoas com algum tipo de deficiência. Estes tipos de software estão dentro de uma área maior chamada Tecnologia Assistiva (TA). Tecnologias assistivas são recursos e serviços que visam facilitar o desenvolvimento de atividades da vida diária para pessoas com deficiência, promovendo maior independência ao permitir que essas pessoas realizem tarefas que antes eram incapazes ou tinham grande dificuldade em realizar. Tecnologias assistivas auxiliam pessoas com dificuldade em falar, digitar, escrever, ver, ouvir, aprender, andar e muitas outras. Diferentes tipos de deficiência requerem diferentes tipos de tecnologias assistivas especializadas. Alguns dos mais conhecidos tipos de tecnologia assistiva incluem: a cadeira de rodas, dispositivo que facilita a locomoção de pessoas incapazes ou com dificuldade de caminhar; o aparelho auditivo, dispositivo que amplifica as ondas sonoras do ambiente com o intuito de corrigir alguma deficiência auditiva do usuário; e os chamados softwares de ampliação de tela, softwares que permitem a ampliação de certas porções da tela para auxiliar pessoas com baixa visão.

No contexto das enfermidades que atingem o uso das mãos, uma das mais conhecidas é a lesão por esforço repetitivo (LER), uma síndrome caracterizada por um grupo de doenças - tendinite, tenossinovite, síndrome do túnel do carpo, dedo em gatilho e etc - que afeta músculos, nervos e tendões dos membros superiores. Além de provocar dor e inflamação, esse distúrbio também pode acarretar no comprometimento da região atingida. A LER é usualmente causada por movimentos repetitivos, tais como: digitação, má postura, tocar piano, dirigir, fazer crochê e etc.

Segundo dados do Anuário Estatístico da Previdência Social, no ano de 2019 a LER figurou entre as 50 causas mais comuns de acidentes de trabalho no Brasil. Dentre essas causas encontram-se a tenossinovite (inflamação ou infecção no tecido que cobre o tendão) e a mononeuropatia dos membros superiores (lesão no nervo periférico), ambas condições que comprometem o uso das mãos.

Indivíduos incapazes ou com dificuldade de interagir com computadores em função da LER podem contar com a ajuda de tecnologias assistivas baseadas em reconhecimento de voz, nesse tipo de tecnologia o usuário utiliza a sua voz, ao invés

das mãos, para interagir diretamente com o computador. Alguns exemplos de tecnologias assistivas baseadas em reconhecimento de voz incluem as assistentes de voz: Siri, Alexa e Cortana. Utilizando a assistente de voz Siri, por exemplo, é possível escrever e enviar um email sem o uso das mãos, utilizando apenas a voz.

Dentre os inúmeros profissionais atingidos pela LER encontram-se os programadores, profissionais cujo trabalho demanda o uso constante das mãos, e portanto, mais suscetíveis a LER. Isso se deve ao fato de que, historicamente, o processo de desenvolvimento de software tem se dado através de repetidas interações entre mouse, teclado e as mãos. Um exemplo disso são os editores de código, onde a principal forma de interação é através do teclado (BEGEL, 2005).

Dito isso, chegamos ao tema central deste trabalho: a criação de uma ferramenta de tecnologia assistiva baseada em reconhecimento de voz capaz de auxiliar programadores com LER a programar.

1.1. OBJETIVOS

A seguir serão listados os objetivos principais a serem alcançados no desenvolvimento deste trabalho.

1.1.1. Objetivo Geral

O objetivo deste trabalho é a criação de uma aplicação capaz de auxiliar programadores a programar em JavaScript sem o uso das mãos. Essa aplicação será capaz de escutar e analisar comandos de voz e então manipular um editor de código qualquer com o intuito de executar esse comando.

Utilizando-se desta aplicação ao invés de fazer o uso das mãos para escrever no editor de código, por exemplo, as instruções necessárias para declarar uma variável, o programador poderá simplesmente expressar em voz alta que deseja declarar uma variável que, em seguida, a aplicação se conectará automaticamente ao editor de código com a intenção de escrever as instruções necessárias para declarar uma variável na linguagem JavaScript.

Para isso, a aplicação deverá ser capaz de: captar comandos de voz através

do microfone do computador, transformar o comando de voz em texto, validar o comando e, por fim, conectar-se ao editor de código com o intuito de executar o comando.

1.1.2. Objetivo Sumarizado

- Implementar uma funcionalidade capaz de fazer reconhecimento de voz em tempo real. Essa funcionalidade transformará comandos de voz em texto.

- Definir uma linguagem e desenvolver um mecanismo para reconhecer tal linguagem utilizando-se de autômatos finitos. O intuito dessa linguagem é representar todos os comandos de voz disponíveis em forma textual.

- Desenvolver uma ferramenta capaz de interagir com editores de código. Essa ferramenta será capaz de manipular um editor de código para, por exemplo, remover uma linha.

1.2. ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado em 5 capítulos fundamentais que expõem integralmente o conteúdo do mesmo. Os capítulos são:

- **Capítulo 1 – Introdução:**
  O presente capítulo. Visa apresentar de forma sucinta o tema central, a motivação, a organização e os objetivos deste trabalho.

- **Capítulo 2 – Fundamentação Teórica:**
  Neste capítulo serão apresentados as ferramentas e os conceitos utilizados no decorrer do desenvolvimento do trabalho. Serão apresentados os conceitos de autômato finito determinístico, *STT Services* (*speech-to-text services*), *string distance*, automação de interface de usuário e etc.

- **Capítulo 3 – Desenvolvimento da aplicação Speech2Code:**

Neste capítulo será mostrado e documentado como todos os conceitos e tecnologias descritas no capítulo 2 interagem entre si para alcançar os objetivos propostos deste trabalho.

- **Capítulo 4 – Resultados:**
  Além de um passo a passo demonstrando como utilizar a aplicação, neste capítulo também será avaliado quais conjuntos de comandos básicos da linguagem de programação JavaScript foram implementados.

- **Capítulo 5 – Conclusão:**
  Neste capítulo é apresentado um resumo do trabalho, seus objetivos, a que grau estes objetivos foram alcançados e sugestões de melhorias futuras.

2. FUNDAMENTAÇÃO TEÓRICA

A seguir serão apresentados os conceitos e tecnologias utilizados ao longo do desenvolvimento deste projeto.

2.1. RECONHECIMENTO E ANÁLISE DE COMANDOS VOZ

Um comando é uma frase que está associada diretamente a uma ação que envolva a manipulação do editor de código. Por exemplo, a frase “Vá para a linha 2” pode ser caracterizada como um comando para trocar de linha cuja ação é manipular o editor de código para que a linha seja trocada. A aplicação de tecnologia assistiva proposta neste trabalho funciona ouvindo comandos de voz do usuário e então realizando ações associadas a estes comandos.

A seguir serão listados todos os conceitos e tecnologias envolvidas na tarefa de transformar uma frase dita em voz alta em texto, analisar esse texto com o intuito de caracterizá-lo como um comando válido e extrair informações importantes para a realização de tal comando.

2.1.1. RECONHECIMENTO DE FALA

Reconhecimento de fala, também conhecido como reconhecimento automático de fala é uma área interdisciplinar da linguística computacional que tem como objetivo o desenvolvimento de métodos e tecnologias que permitem a transformação da linguagem falada em linguagem textual por computadores. Embora o termo reconhecimento de voz seja frequentemente utilizado como sinônimo para reconhecimento de fala, este último foca em transformar a linguagem falada em texto, enquanto o primeiro foca na identificação individual de vozes de pessoas (IBM Cloud, 2020).

Além das peculiaridades da linguagem humana, os problemas de natureza interdisciplinar tornam o reconhecimento da fala uma das áreas mais difíceis da ciência da computação. Dentre as disciplinas envolvidas na resolução de problemas desta área encontram-se (RABINER, Lawrence; JUANG Biing-Hwang, 1993):

- Processamento de sinal: O processo de extrair informações relevantes do sinal acústico gerado pela voz de forma eficiente e robusta.

- Física (Acústica): A ciência da relação entre a manifestação física da fala (sinal gerado pela voz) o mecanismo fisiológico que produz a fala (o sistema vocal humano) e como a fala é percebida (o sistema auditivo humano).

- Reconhecimento de Padrões: O conjunto de algoritmos usados para identificar e classificar padrões de agrupamento em uma base de dados.

- Linguística: A relação entre sons (fonologia), palavras em uma linguagem (sintaxe), significado das palavras (semântica) e o sentido derivado do contexto (pragmática).

Um reconhecedor de fala tem como entrada um sinal de fala obtido a partir de um transdutor e a partir desse sinal é realizado um mapeamento a fim de descobrir a palavra falada, ou seja, transcrever o que foi falado. O processo de reconhecimento é dividido em quatro etapas, aquisição do sinal de voz, pré-processamento, extração de informações e a última que pode ser a geração dos padrões de voz, quando na fase de treinamento ou a classificação, quando na fase de reconhecimento, que

utiliza os padrões de voz gerados na fase de treinamento (Silva, 2009).

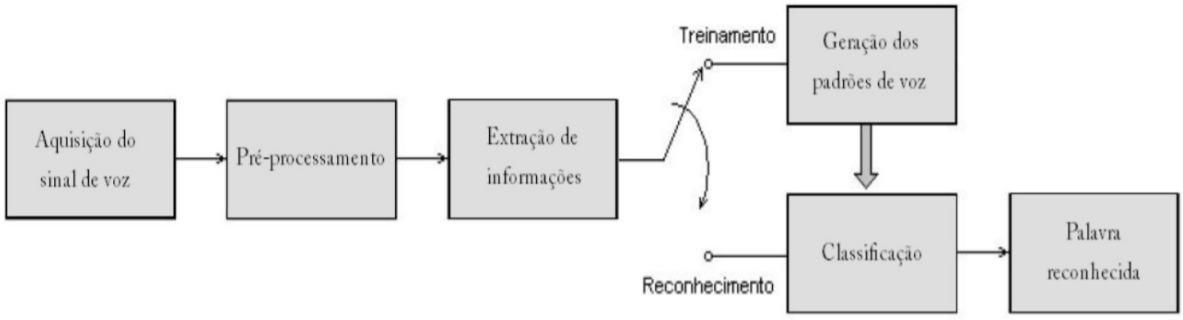

Figura 1. Processos de um sistema de reconhecimento de fala. Fonte: Silva (2009)

Um reconhecedor de fala é avaliado com base na sua taxa de precisão, ou seja, taxa de erro no reconhecimento de palavras e velocidade de reconhecimento. Vários fatores podem afetar a taxa de erro, tais como a pronúncia, sotaque, tom, volume e ruído. Reconhecedores de fala tem como objetivo alcançar uma taxa de erro similar a de dois humanos conversando (IBM Cloud, 2020).

Sistemas de reconhecimento de fala têm aplicações em diversas áreas. Na realidade qualquer atividade que envolva interação humano-máquina pode potencialmente utilizar estes sistemas. Atualmente várias aplicações já estão sendo concebidas com um sistema de reconhecimento de fala incorporado. Dentre as aplicações mais comuns encontra-se: sistemas de controle e comando, sistemas de telefonia, sistemas de transcrição, centrais de atendimento ao cliente, assistentes virtuais e segurança (Silva, 2009).

2.1.1.1. SOFTWARES DE RECONHECIMENTO DE FALA

Em poucas palavras, *speech to text software (STT)*, ou *automatic speech recognition software (ASR)*, ou *voice to text software*, é um programa de computador que usa algoritmos linguísticos para classificar e transformar sinais sonoros em texto.

Softwares de reconhecimento de voz podem ser usados em casos onde o indivíduo precisa gerar um grande volume de conteúdo textual sem a necessidade

de escrever manualmente. Essa tecnologia também é usada por indivíduos incapazes ou com grande dificuldade de utilizar um teclado.

Dentre os diversos produtos no mercado para fazer reconhecimento de voz, destacam-se: *Cloud Speech-to-text* (Google), *Azure Speech to Text* (Microsoft) e o *Watson Speech to Text* (IBM), todos são serviços baseados em nuvem capazes de fazer reconhecimento de voz em múltiplos idiomas e prometem alta taxa de acurácia no reconhecimento de palavras. Por se tratarem de produtos voltados ao uso geral, não limitados a um domínio específico, tendem a ser de fácil uso e baixo custo.

Para a realização deste trabalho foram consideradas as ferramentas de reconhecimento de voz da Google e Microsoft. Embora ambas funcionem de forma similar, por meio de web APIs, e tenham suporte às mesmas funcionalidades, a ferramenta da Microsoft apresentou um grau de acurácia maior no reconhecimento de palavras e possui uma documentação mais abrangente. Dito isto, neste projeto usaremos como ferramenta de reconhecimento de fala o *Azure Speech To Text*.

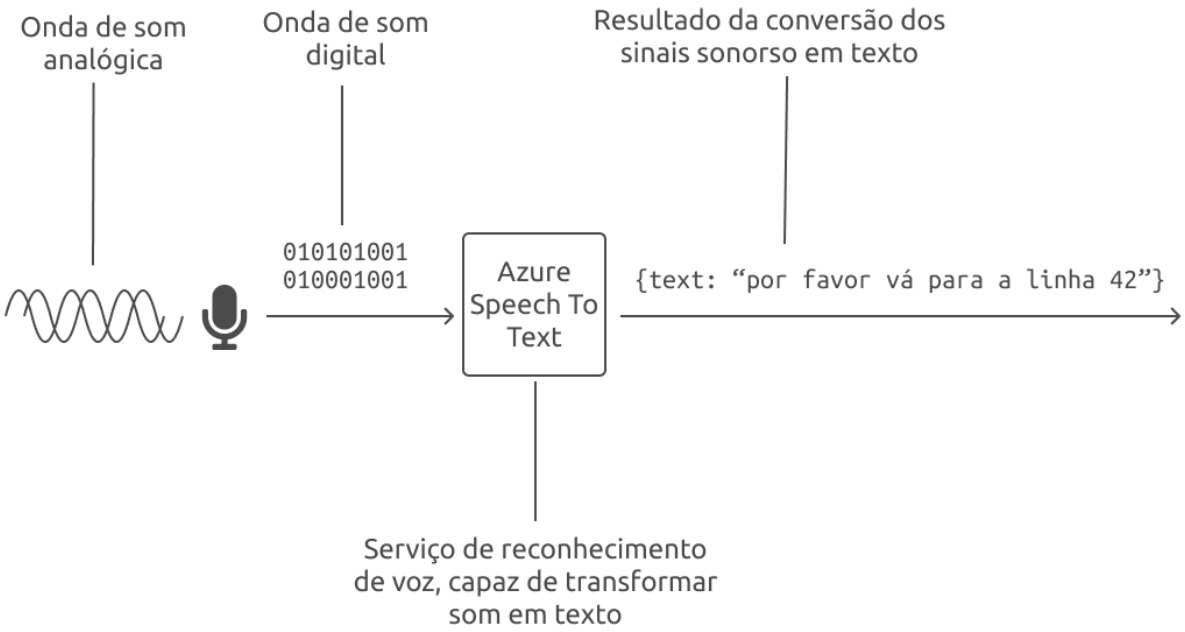

Figura 2. Funcionamento simplificado de um serviço de reconhecimento de voz. Fonte: O autor

2.1.1.1.1. AZURE SPEECH TO TEXT

O *Microsoft Azure Speech To Text*, software comercial de reconhecimento de

voz da Microsoft, permite a criação de aplicativos com capacidade de reconhecimento de fala de forma simples e rápida. Possui suporte a mais de 85 idiomas diferentes, alta taxa de acurácia, suporte a diferentes linguagens de programação e personalização de modelos. Ele também oferece a conversão de áudio em texto a partir de fontes diferentes, como microfones, arquivos de áudio e *blobs*.

Por se tratar de um *SaaS* (*Software as Service*) a primeira coisa a se fazer antes de usar o serviço é criar uma conta no Microsoft Azure, plataforma de computação em nuvem da Microsoft. Com essa conta é possível contratar uma ampla gama de serviços e produtos voltados para os mais diferentes tópicos como armazenamento de dados, segurança, internet das coisas, inteligência artificial e etc. A Azure oferece vários tipos de contas diferentes para diferentes perfis, variando dos mais básicos aos mais robustos, incluindo uma conta gratuita. No contexto desse projeto um dos benefícios da conta gratuita são 5 horas por mês de conversão de fala em texto grátis, ou seja, é possível realizar até 5 horas de reconhecimento de fala por mês sem pagar nada por isso.

O uso do *Azure Speech To Text* se dá através de um Web Service disponível a usuários com uma conta no Microsoft Azure. Esse Web Service age como um intermediário para o recurso de reconhecimento fala. Tem como função receber e validar requisições de reconhecimento de fala e então responder de forma apropriada. Utilizando esse Web Service é possível, por exemplo, enviar uma requisição para realizar reconhecimento de fala em arquivo de áudio e receber, como resposta, a transcrição em texto desse arquivo.

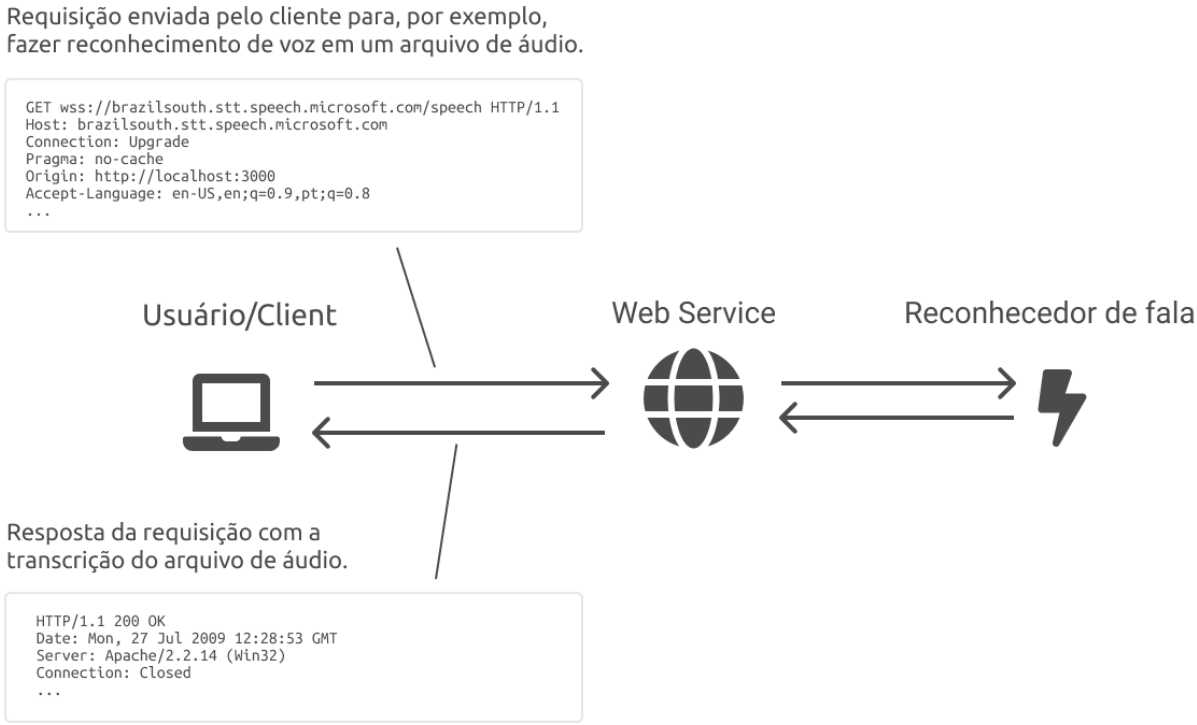

Figura 3. Exemplo da interação entre o cliente, Web Service e reconhecedor de fala. Fonte: O autor

2.1.1.1.1.1. SpeechSDK

A Azure oferece uma *SDK (*Kit de Desenvolvimento de Software*)* chamada *SpeechSDK* que visa facilitar a interação com o Web Service, essa SDK abstrai e encapsula toda a complexidade do processo de comunicação com o Web Service. A SDK está disponível em diversas linguagens de programação incluindo Java, JavaScript, Python e etc. Neste projeto foi utilizada a sua implementação em JavaScript, por se tratar de um projeto voltado para a Web. Fazendo uso da SDK é possível realizar o processo descrito na figura 3 em poucas linhas de código como mostrado abaixo:

```
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)
})
```

Figura 4. Uso da SpeechSDK para realizar a transcrição de um arquivo de áudio. Fonte: O autor

A SDK oferece 3 tipos básicos de reconhecimento de fala todos eles podem ser feitos a partir de um microfone, arquivo ou *blob*:

- Reconhecimento Pontual - O reconhecimento é feito em todo o discurso até que o usuário pare de falar. “Parar de falar” é considerado como uma pausa longa no discurso. Esse tipo de reconhecimento de fala é indicado para comandos e perguntas, casos em que após o usuário parar de falar o processo de reconhecimento deve se encerrar de forma automática.

- Reconhecimento Contínuo - O reconhecimento é feito de forma contínua em todo o discurso até que o usuário explicitamente o pare. É indicado em casos onde o processo de reconhecimento deve continuar mesmo em longos períodos de silêncio. Exemplos de uso incluem monólogos e conversas entre duas ou mais pessoas.

- Reconhecimento de palavra chave - O reconhecimento só começa quando o usuário diz uma palavra chave e só acaba quando o usuário o pare de forma explícita. Nesse tipo de reconhecimento a fala é sempre gravada, analisada e os resultados descartados, a não ser que a palavra chave seja dita, então os resultados não são descartados mas sim entregues ao usuário. Exemplos desse tipo de reconhecimento de fala incluem o comando “Ok Google, …” (onde “Ok Google” é a palavra chave) da assistente de voz *Google Assistant*.

2.1.1.1.1.2. Pontuação Automática

Outra funcionalidade importante do *Azure Speech To Text* para o contexto desse projeto é a pontuação automática no reconhecimento de sentenças. Essa funcionalidade é capaz de analisar uma sentença falada e inferir onde, de acordo com as regras da gramática do idioma, são necessários sinais de pontuação. Com essa funcionalidade ativada ao dizer a frase “quem é você” em um tom de pergunta e então, realizar o processo de reconhecimento de fala o resultado obtido seria a frase “quem é você ?” com o sinal de interrogação.

Ao desativar essa funcionalidade a inclusão de pontuação na transcrição da fala precisa ser feita de forma explícita. Ao dizer a frase “Amarelo Azul e Verde” o resultado não será “Amarelo, Azul e Verde”, para alcançar tal resultado é preciso dizer “Amarelo vírgula Azul e Verde”, ou seja, a vírgula - pontuação - precisa estar explícita na sentença. Essa diferença é de extrema importância para este projeto uma vez que os sinais de pontuação tem um significado importante na programação.

2.1.2. STOP WORDS

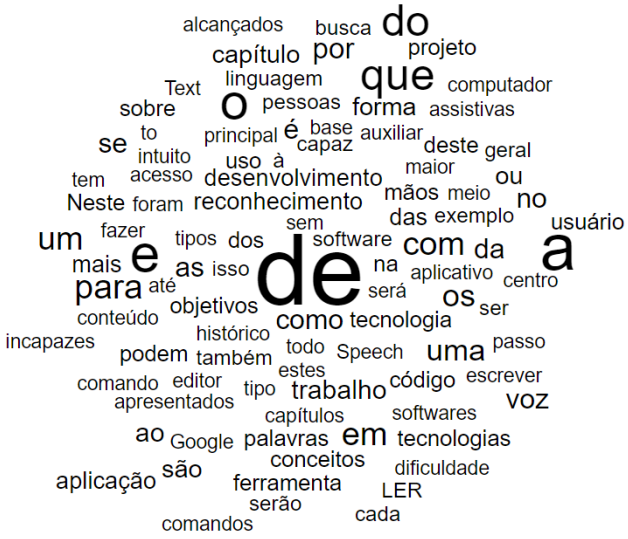

Figura 5. Nuvem das palavras mais comuns deste documento. Fonte: O autor

No contexto de processamento de linguagem natural, *stop words* ou palavras de parada são palavras que podem ser removidas antes ou após o processamento de documentos. *Stop words* são usualmente classificadas como as palavras mais comuns de uma linguagem, com altas chances de ocorrerem em qualquer documento, e que tendem a contribuir mais para a conformidade gramatical e sintática da linguagem do que para o real significado do documento (Wilbur; Sirotkin, 1992). Não existe uma lista definida de quais palavras são classificadas como *stop words* e elas podem variar de aplicações para aplicações.

Na sentença “Vá para a linha número 42” as palavras “para” e “a” podem ser classificadas como *stop words* e removidas sem a perda significativa no sentido da sentença, isso se deve ao fato de que as palavras “para” e “a” só existem para completar o verbo indireto “vá”.

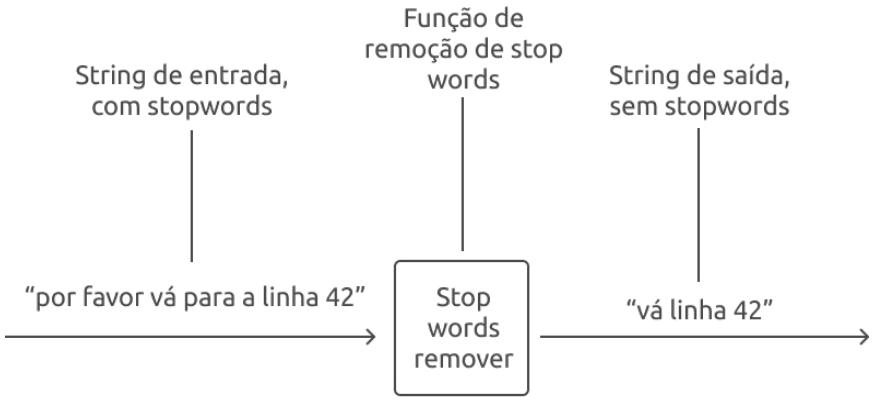

Figura 6. Funcionamento simplificado de uma função de remoção de stop words. Fonte: O autor

Neste projeto *stopwords* serão removidas para simplificar o processo de reconhecimento de comandos. Imagine que o usuário tenha dito em voz alta uma frase e, após feito o reconhecimento de fala, fosse constatado que ele disse a frase “por favor declare uma variável chamada valor igual ao número 42”. A próxima fase seria verificar se essa frase corresponde a um comando válido ou não, porém como visto anteriormente *stopwords* são palavras que não influenciam no significado da sentença, portanto podemos removê-las. Logo, a sentença enviada para a próxima fase seria “~~por favor~~ declare ~~uma~~ variável chamada valor igual ~~ao~~ número 42”, efetivamente removendo as palavras de parada “por favor”, “um” e “ao” da sentença

original.

2.1.3. TEORIA DOS AUTÔMATOS

A Teoria dos Autômatos é um ramo teórico da ciência da computação que tem suas raízes no século 20, quando matemáticos começaram a desenvolver - tanto em teoria quanto na prática - máquinas que imitavam certas funções do cérebro humano (Hopcroft; Motwani; Ullman, 1979). A própria palavra autômato, intimamente relacionada com a palavra "automação", denota processos automáticos que realizam uma série de operações pré determinadas de forma automática. Simplificando, a teoria dos autômatos lida com a lógica de computação de máquinas simples, chamadas de autômatos. Por meio de autômatos, os cientistas da computação são capazes de entender como as máquinas executam funções e resolvem problemas e, mais importante, o que significa uma função ser computável ou uma questão decidível.

Autômatos são modelos abstratos de máquinas que fazem a análise de uma cadeia de caracteres movendo-se por uma série de estados. Em cada estado, uma função de transição determina o próximo estado com base em uma parte da cadeia de caracteres da entrada. Como resultado, uma vez que o autômato atinge um estado de aceitação, é dito que ele aceita essa entrada.

O principal objetivo da teoria dos autômatos é desenvolver métodos pelos quais os cientistas da computação possam descrever e analisar o comportamento dinâmico de sistemas discretos. O comportamento desses sistemas discretos é determinado pela maneira como o sistema é construído a partir de armazenamento e elementos combinacionais (Eric Roberts, 2009). As características de tais máquinas incluem:

- Entrada: sequência de símbolos selecionados de um conjunto finito de sinais de entrada.
- Saída: sequência de símbolos selecionados de um conjunto finito M. Onde M é o conjunto {a1, b2, b3 ... bn} onde n é o número total de saídas.
- Estados: conjunto finito Q, cuja definição depende do tipo de autômato.

Existem três famílias principais de autômatos: máquina de estados finita, autômatos de pilha e máquinas de Turing. Essas famílias de autômatos podem ser interpretadas de forma hierárquica, onde a máquina de estado finita é o autômato mais simples e a máquina de Turing é a mais complexa. O foco deste projeto está na máquina de estado finita e no autômato de pilha.

2.1.3.1. Autômato Finito Determinístico

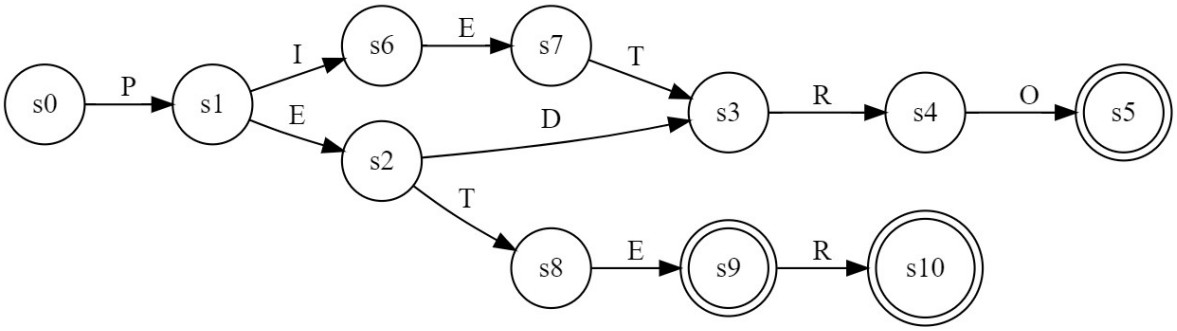

Figura 7. Autômato capaz de reconhecer algumas variações do nome Pedro. Fonte: O autor

Autômato finito determinístico (AFD) ou máquina de estados finita determinística é uma máquina de estado que ao percorrer uma sequência única de estados aceita ou rejeita uma cadeia de símbolos (Hopcroft; Motwani; Ullman, 1979). No autômato da figura acima os estados são representados por círculos, os estados finais por círculos duplos, as funções de transição por setas e o estado inicial por *s0*. É dito que um autômato aceita uma cadeia de símbolos, se ao consumir todos os símbolos desta cadeia ele encontra-se em um estado final, caso contrário ele a rejeita.

Formalmente um Autômato Finito Determinístico A é uma quíntupla, (Q, Σ, δ, q0, F) onde:

- Σ é o conjunto finito de símbolos chamado alfabeto
- Q é o conjunto finito de estados
- δ é a função de transição
- q0 é o estado inicial (q0 ∈ Q)

- F é o conjunto de estados de aceitação (F ⊆ Q)

Na teoria dos autômatos uma máquina de estados finita só é considerada um autômato finito determinístico se:

- Cada uma de suas transições é determinada apenas pelo estado de origem e o símbolo de entrada.
- É necessário ler um símbolo de entrada para uma mudança de estado.

Caso a máquina de estados finita não atenda a essas condições ela é chamada de autômato finito não determinístico.

O AFD apresentado na figura abaixo reconhece a cadeia de símbolos “PIETRO” ao percorrer os estados: *s0*, *s1*, *s6*, *s7*, *s3*, *s4* e *s5*, sendo *s0* e *s5* estados inicial e final, respectivamente.

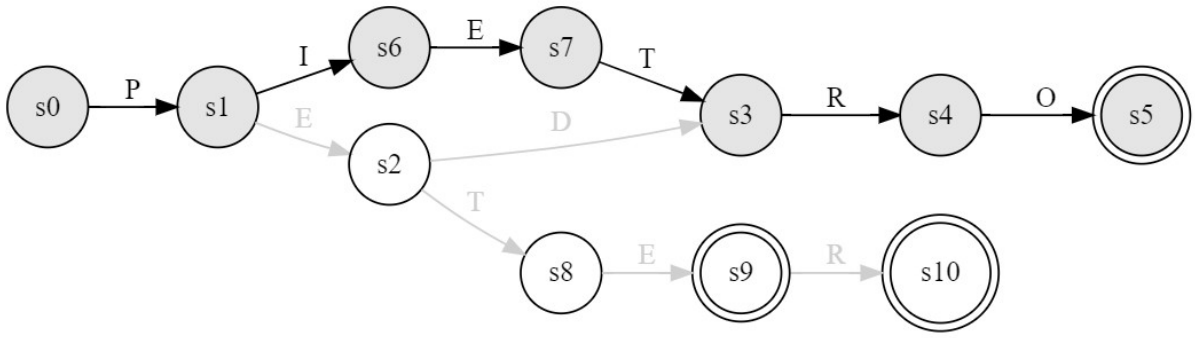

Figura 8. Autômato com a sequência de estados a percorrer para aceitar a cadeia de símbolos “PIETRO” em realce. Fonte: Autor

AFDs podem ser usados na resolução de problemas que envolvem linguagens regulares como expressões regulares e na análise léxica de um compilador. Por exemplo, é possível modelar um autômato capaz de dizer se um endereço de email é válido ou não (GOUDA; Prabhakar, 2009).

2.1.3.2. Autômato De Pilha

O autômato de pilha é essencialmente um autômato finito não determinístico,

que aceita transições vazias (λ), com a adição de mais uma funcionalidade: uma pilha na qual ele pode guardar um série de símbolos. A presença dessa pilha significa que, ao contrário do autômato finito determinístico, o autômato de pilha pode guardar uma quantidade infinita de informação. Contudo, diferente de um computador de propósito geral, que também pode guardar um grande volume de informação, o autômato de pilha só pode acessar a informação em sua memória/pilha em um modelo *last-in-first-out (LIFO)*, característico da estrutura de dados pilha.

Em função dessa limitação existem linguagens que podem ser reconhecidas por um computador de propósito geral, mas não por autômatos de pilha. Autômatos de pilha por definição só reconhecem linguagens livres de contexto (Hopcroft; Motwani; Ullman, 1979). Segundo a Hierarquia de Chomsky os autômatos de pilha são modelos de computação equivalentes às gramáticas livres de contexto.

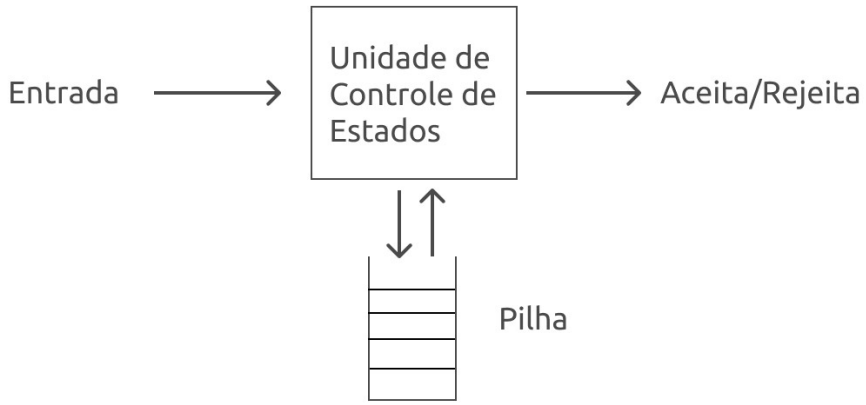

Figura 9. Autômato de pilha representado como uma máquina de estados finita com acesso a uma pilha. Fonte: Adaptado de Hopcroft; Motwani; Ullman, 1979.

O dispositivo mostrado na figura acima exemplifica o funcionamento de um autômato de pilha. Conforme a “unidade de controle de estados” lê os símbolos da entrada, o autômato de pilha pode basear a sua mudança de estado em seu estado atual, o símbolo no topo da pilha e o símbolo lido. A mudança de estado pode ocorrer ainda de forma espontânea sem que seja lido um símbolo da entrada, essa transição espontânea é chamada de transição vazia ou transição *epsilon (λ)*. Segundo Hopcroft, Motwani e Ullman (1979) em cada transição um autômato de

pilha:

1. Lê um símbolo da entrada que será usado para fazer a transição, caso seja uma transição vazia nenhum símbolo é lido (*λ*).

2. Vai para um novo estado que pode ser o mesmo do estado anterior.

3. Troca o símbolo no topo da pilha por outro símbolo. Esse outro símbolo pode ser uma string vazia (λ), o que corresponde a remoção do símbolo no topo da pilha. O mesmo símbolo que já encontra-se no no topo da pilha, o que corresponde a nenhuma alteração na pilha. Um outro símbolo qualquer, o que corresponde à substituição do topo da pilha, sem que seja feita uma remoção ou adição. E por fim o topo da pilha pode ser substituído por dois ou mais símbolos, o que pode ter o efeito de substituir o topo da pilha e então adicionar mais símbolos a ela.

2.1.3.3. RECONHECENDO FRASES USANDO AUTÔMATOS DE PILHA

Como vimos anteriormente, um comando é uma frase associada a uma ação que envolve manipular um editor de código. Dada uma frase em formato textual, o problema de decidir se essa frase é ou não um comando válido pode ser resolvido com o auxílio de autômatos de pilha onde o alfabeto de entrada e as funções de transições são baseados em palavras. A ideia é que um autômato de pilha, capaz de reconhecer uma ou mais frases, esteja associado a um comando. Considere o comando A que está associado ao autômato B que reconhece um conjunto de frases K, considere também uma frase qualquer x. A ação relacionada ao comando A só será executada caso a frase x pertença a K.

Ao utilizar um autômato de pilha para reconhecer uma frase, os símbolos de entrada que compõem o alfabeto podem ser tratados como as palavras individuais que compõem a frase. Um autômato de pilha A reconhece a frase “vá para a linha 42” se ao percorrer todas as palavras dessa frase (vá, para, a, linha, 42) ele encontra-se em um estado final. A extração das palavras que compõem a frase é feita utilizando um *tokenizer* simples.

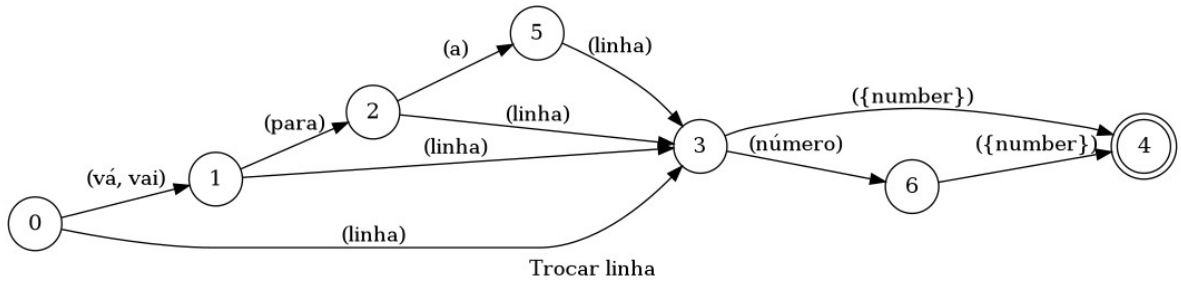

Figura 10. Autômato usado para reconhecer o comando para trocar de linha. Fonte: O autor

O autômato de pilha da figura acima é responsável por reconhecer frases relacionadas ao comando de trocar de linha. Como dito anteriormente, todas as funções de transição são baseadas em palavras e não em símbolos, ainda na figura 10 podemos observar que a transição do estado 0 para o estado 1 só ocorre caso a palavra de entrada seja “vá” ou “vai”, a transição do estado 1 para o estado 2 somente quando a palavra de entrada é igual a “para” e assim sucessivamente. Outra peculiaridade é a notação “*({number})*” usada na transição entre os estados 3 e 4, essa notação denota que a transição deve aceitar qualquer palavra lida e guardá-la no topo da pilha.

Ao todo o autômato de pilha da figura 10 pode reconhecer 8 frases relacionadas ao comando de trocar de linha, a seguir veremos como ele se comporta ao tentar reconhecer a frase “vá para linha 42” que após passar pelo *tokenizer* resulta na sequência de entrada {“vá”, “para”, “linha”, “42”}:

1. No estado inicial 0 é lida a primeira palavra da entrada, “vá”. O estado 0 possui transições para os estados 3 e 1, porém somente a função de transição para o estado 1 aceita a palavra “vá”, então o autômato consome a palavra de entrada e vai para o estado 1.

2. No estado 1 é lida a próxima palavra da sequência de entrada, “para”. O estado 1 possui transições para os estados 2 e 3, porém somente a função de transição para o estado 2 aceita a palavra “para”, então o autômato consome a palavra de entrada e vai para o estado 2.

3. No estado 2 é lida a próxima palavra da sequência de entrada, “linha”. O estado 2 possui transições para os estados 3 e 5, porém somente a função de transição para o estado 3 aceita a palavra “linha”, então o autômato consome a palavra de entrada e vai para o estado 3.

4. No estado 3 é lida a próxima palavra da sequência de entrada, “42”. O estado 3 possui transições para os estados 4 e 6. A função de transição para o estado 6 requer a palavra “número” que é diferente da palavra “42”. Já a função de transição para o estado 4 aceita qualquer palavra com a condição de que ela seja escrita na pilha. Nesse caso o autômato consome a palavra de entrada e a adiciona no topo da pilha, por fim ele vai para o estado 4.

5. No estado 4 não existem mais palavras na sequência de entrada e o autômato se encontra em um estado final. É dito então que o autômato identifica a frase “vá para linha 42” como sendo um comando válido para trocar de linha e sua pilha, que atualmente contém a palavra “42”, pode ser usada como um argumento ao executar a ação relacionada a esse comando.

Como demonstrado, o autômato da figura 6 reconhece a frase “vá para linha 42” como uma frase válida para o comando de trocar de linha, ele passa pelos estados 0, 1, 2, 3, 4 e termina com a pilha contendo a palavra “42”. Os elementos dessa pilha podem ser usados como argumentos para executar a ação associada ao comando, no caso apresentado, a ação é trocar de linha e o argumento é a linha número 42.

Dito isso, dado um conjunto M de comandos, decidir se uma frase x é um comando válido ou não, é apenas uma questão de iterar sobre cada elemento de M e utilizar seu autômato associado para decidir se a frase pertence ou não ao comando.

2.1.3.3.1. FUNÇÕES DE TRANSIÇÃO

Outra mudança significativa feita nos autômatos de pilha com o intuito de

diminuir a sua complexidade foi a definição de 3 tipos de função de transição. Lembre-se que a função de transição em um autômato de pilha decide se o autômato pode ir para o próximo estado com base no símbolo de entrada, o estado atual e o topo da pilha (Hopcroft; Motwani; Ullman, 1979).

2.1.3.3.1.1. TRANSIÇÃO POR SIMILARIDADE

Normalmente a função de transição compara o símbolo lido, com o símbolo necessário para ir para o próximo estado, caso eles sejam iguais a transição é feita. Na transição por similaridade esse processo é alterado para que a transição também seja feita caso os símbolos sejam similares.

Utilizando esse tipo de transição é possível filtrar variações verbais em palavras e erros cometidos pelo processo de reconhecimento de voz. Considere o autômato abaixo, ele aceita tanto as frases “declarar uma” tanto como “declare uma”, isso se deve ao fato de que embora as palavras “declarar” e “declare” sejam diferentes elas ainda são muito similares.

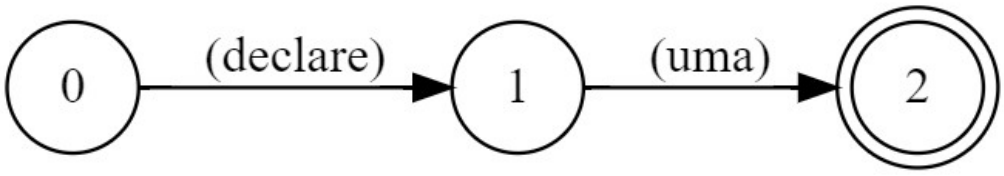

Figura 11. Transição por similaridade. Fonte: O autor

A transição por similaridade é a padrão e é recomendada para todas as transições envolvendo palavras.

A similaridade entre duas palavras é determinada através de um algoritmo que mede a distância entre duas strings (cadeia de caracteres também chamada de palavra).

2.1.3.3.1.1.1. String Distance

String distance ou *string metric* é uma métrica que mede o grau de

similaridade entre dois fragmentos de texto, essa métrica é amplamente utilizada em problemas como a busca aproximada de texto e o corretor ortográfico. Por exemplo, as palavras “Emergir” e “Imergir” podem ser consideradas similares (Lu; et al, 2013).

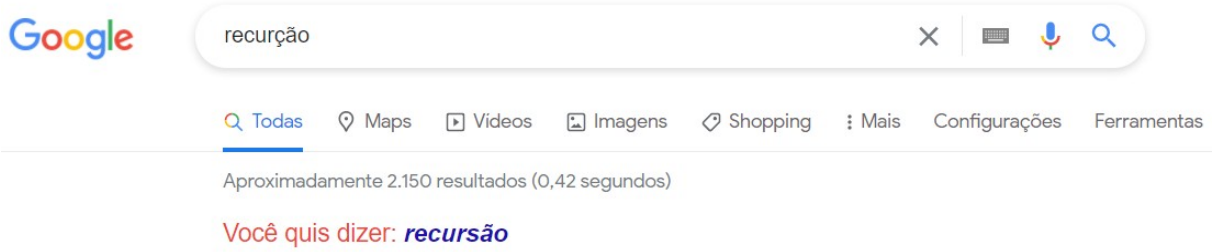

Figura 12. Exemplo do emprego de string distance no corretor ortográfico. Fonte: *print screen* do website Google.com

Dentre os diversos algoritmos para calcular a similaridade entre *strings* (cadeias de caracteres) o mais popular é a chamada distância de Levenshtein, que mede o número de edições (inserção, deleção ou substituição) para transformar uma string em outra. Por exemplo, a distância Levenshtein entre as palavras inglesas "kitten" (gato) e "sitting" (sentando-se) é 3, já que com apenas 3 edições conseguimos transformar uma palavra na outra.

Neste projeto a métrica *string distance* será usada como base para a função de transição dos autômatos responsáveis por reconhecer e analisar comandos. O emprego desta métrica torna o autômato mais tolerante a erros de ortografia e variações verbais.

2.1.3.3.1.2. Transição por Classe

A transição é feita quando a palavra de entrada pertence a uma classe predeterminada de palavras. Essa transição é usada em casos onde não é possível enumerar todas as palavras aceitas por uma transição, portanto elas são agrupadas sobre uma classe X e diz se que a transição ocorre para qualquer palavra que pertença ao grupo X. Transições de classe existem com o intuito de simplificar a representação e compartilhar lógica entre autômatos diferentes.

O autômato da figura abaixo é capaz de reconhecer as frases “número 1”,

“número 2”, “número 3”, “número 4” e etc. Ele é capaz de reconhecer qualquer frase no formato “número *$X"* onde *$X* é um número natural qualquer. Isso se deve ao fato de que a transição do estado 2 para o 3 é uma transição de classe, logo essa transição irá para o próximo estado ao ler qualquer palavra que pertença a classe *{number}*.

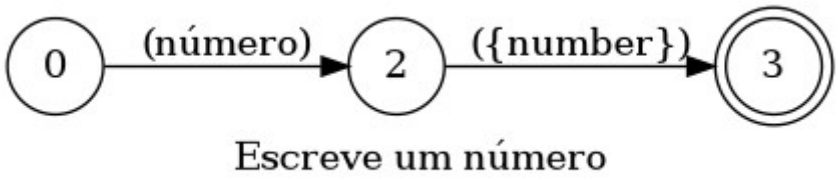

Figura 13. Autômato responsável por reconhecer o comando para escrever um número. Fonte: O autor

Internamente a classe *{number}* é definida como uma expressão regular que aceita qualquer número natural. Outras classes também incluem: *{char}*, que aceita somente um caractere, *{term}*, que aceita qualquer palavra e *{numeral}* que aceita números ordinários.

Transições de classe devem sempre ser denotadas entre chaves do contrário elas serão interpretadas como uma transição de similaridade. Na figura 13 a transição do estado 0 para o estado 2 é uma transição de similaridade enquanto que a transição do estado 2 para o 3 é uma transição de classe, isso se deve ao fato de que o texto desta última encontra-se entre as chaves.

2.1.3.3.1.3. Transição por Autômato

Neste tipo de transição o controle sobre sequência de entrada é passado há um segundo autômato e a transição de estado só ocorre caso esse segundo autômato termine em um estado final. Essencialmente esse tipo de transição permite a composição e a reutilização de autômatos.

Na figura abaixo é possível observar que os autômatos B e C utilizam o autômato A para compor suas funções de transição. As setas em vermelho indicam

a qual autômato a função de transição faz referência.

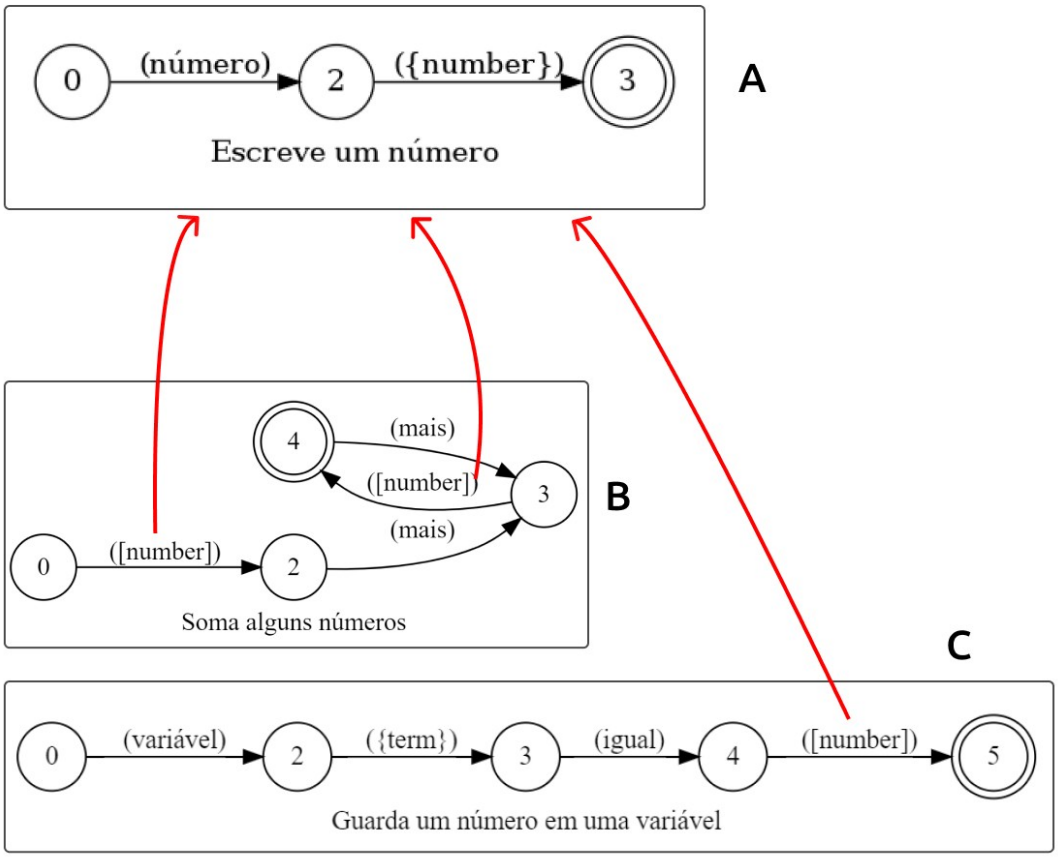

Figura 14. Mostrando como é possível fazer a composição entre autômatos. Fonte: O autor

No autômato B para fazer a transição entre os estados 0 e 2 o conteúdo atual da sequência de entrada é aplicado ao autômato A, caso este termine em um estado final, o autômato B pode prosseguir para o estado 2. Vale notar que o autômato A também pode consumir palavras da entrada e manipular a pilha.

Transições de autômato devem sempre ser denotadas entre colchetes, do contrário elas serão interpretadas como uma transição de similaridade. No autômato C da figura 14 a transição do estado 0 para o estado 2 é uma transição de similaridade enquanto que a transição do estado 4 para o 5 é uma transição de autômato, isso se deve ao fato de que o texto desta última encontra-se entre as colchetes.

2.1.3.4. Representando Autômatos Em Forma Textual

DOT é uma linguagem para descrever grafos em forma textual que depois podem ser transformados em diagramas. Utilizando a *DOT* é possível representar grafos dirigidos, não dirigidos e até mesmo autômatos (Gansner; Koutsofios; North, 2015).

Como já mencionado anteriormente, autômatos de pilha serão uma parte importante deste projeto e a linguagem *DOT* apresentada aqui será a forma padrão de representá-los.

```
1   digraph AssignValue {
2       label="Guarda um número em uma variável";
3
4       0 -> 2 [label="(variável)"];
5       2 -> 3 [label="({term})", store=nomeVariavel];
6       3 -> 4 [label="(igual)"];
7       4 -> 5 [label="([number])", store=valorVariavel];
8   }
```

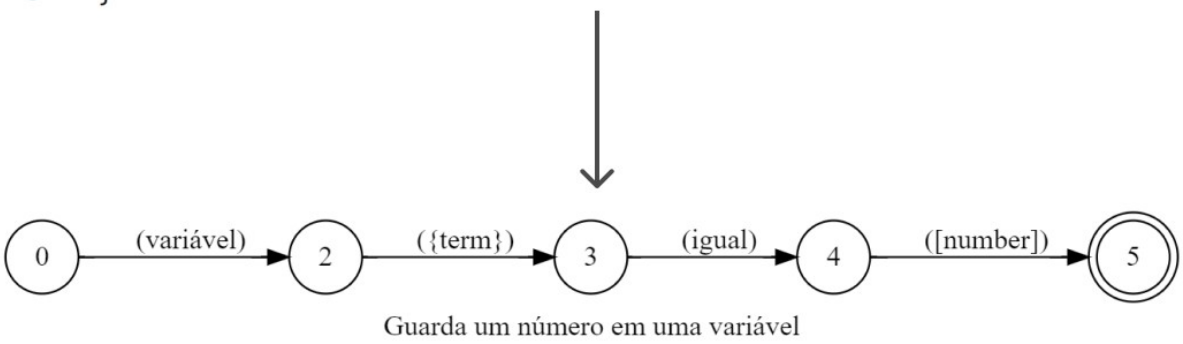

Figura 15. Na parte superior da imagem é possível ver a representação textual, utilizando a linguagem DOT, do autômato de pilha e na parte inferior o mesmo autômato em formato visual.
Fonte: O autor

Na figura 15 é possível observar os três tipos de transições já mencionados:

- 0 → 2: Transição por similaridade, aceita qualquer palavra similar à palavra “variável”.

- 2 → 3: Transição por classe, aceita qualquer palavra pertencente a classe “term”.

4 → 5: Transição por autômato, aplica a entrada atual a um autômato chamado “number” e só vai para o próximo estado caso este autômato a aceite.

Ainda na representação textual da figura 15 é possível observar que as transições nas linhas 6 e 7 possuem um atributo chamado "*store*", esse atributo diz ao autômato para guardar a palavra lida no topo da fila.

2.2. AUTOMAÇÃO DE EDITORES DE CÓDIGO

Como explicado anteriormente, o objetivo central deste trabalho é criar uma ferramenta de tecnologia assistiva capaz de ajudar programadores a programar. A ferramenta elimina a necessidade do programador utilizar o teclado para interagir com o editor de código, basta que o programador diga em voz alta o que precisa ser feito e isso será feito de forma automática no editor de código. Essa funcionalidade pode ser dividida em dois grandes problemas: reconhecimento e análise do comando de voz e manipulação do editor de código para execução do comando. Neste subcapítulo exploraremos o segundo problema: Como um programa A pode interagir com um outro programa externo B.

GUI Automation ou Automação da Interface Gráfica de Usuário é o processo de simular ações do mouse e teclado de forma programática a fim de interagir com um programa externo, pode ser usada em leitores de tela, testes automatizados, entrada automatizada de dados, integração entre programas e migração de conteúdo. Em geral, *GUI Automation* é usada para automatizar tarefas repetitivas e tediosas (UIPATH, 2015).

Neste projeto usaremos duas formas de *GUI Automation* para interagir com editores de código: *PyAutoGUI* e Extensões.

2.3.1. PyAutoGUI

PyAutoGUI é um package da linguagem de programação Python que permite o controle do mouse e teclado para automatizar interações com outros programas

(Sweigart, 2017).

Utilizando este package é possível:

- Mover e clicar com o mouse ou escrever em janelas de outras aplicações.
- Controlar o teclado para, por exemplo, preencher formulários.
- Localizar as coordenadas de elementos como botões e imagens na tela.
- Fechar, mover, redimensionar ou maximizar janelas de outras aplicações.

Uma das maiores vantagens do PyAutoGUI é ser capaz de interagir com qualquer aplicação, é possível, por exemplo, usar o mesmo script para escrever na caixa de mensagens do Whatsapp para preencher os formulários do aplicativo da Declaração de Imposto de Renda. Isso se deve ao fato de que para o PyAutoGUI o que se encontra na tela não é o aplicativo X ou Y mas sim uma série de pixels.

No contexto deste projeto essa generalização se torna um tanto problemática, por exemplo, utilizando o PyAutoGUI para interagir com editores de código tarefas simples como trocar o ponteiro para uma linha arbitrária ou indentar o arquivo atual se tornam complicadas, uma vez que o PyAutoGUI não tem o conceito de editor de código ou número de linhas em um arquivo, mas sim o de pixels em uma tela.

Dito isso, PyAutoGUI não será a ferramenta padrão de automação e só será usada em casos onde uma alternativa mais especialista não estiver disponível.

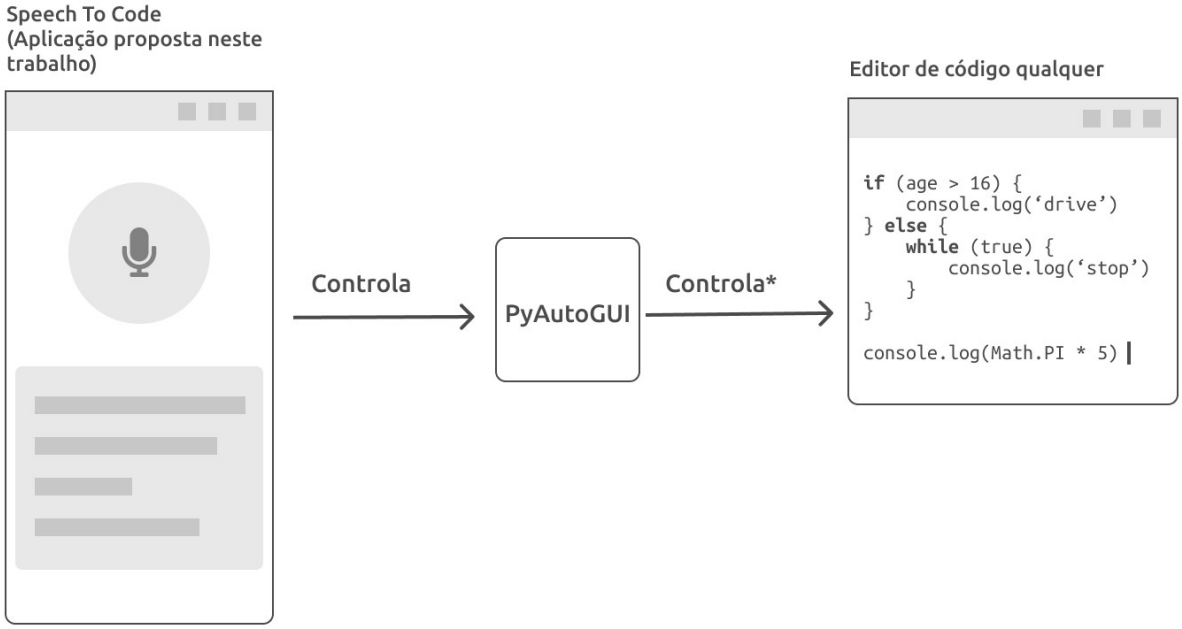

Figura 16. Diagrama mostrando como a ferramenta PyAutoGUI age como um intermediário na comunicação entre programas. Fonte: O autor

2.3.2. EXTENSÕES

Plugins ou extensões são uma funcionalidade importante dos editores de código modernos que permitem que desenvolvedores criem módulos (plugins) capazes de estender ou adicionar novas funcionalidades ao editor de código. Os plugins são executados no contexto do editor de código e por isso tem acesso a APIs internas, que o permitem controlá-lo de forma programática. Plugins podem ser usados para adicionar funcionalidades como: realce de sintaxe para uma linguagem de programação, sugestão e complementação de código, fechamento automático de chaves, parênteses e colchetes e etc.

Dentre os editores de código e ambientes de desenvolvimento integrado com suporte a plugins e extensões encontram-se: *Visual Studio Code*, *Sublime Text*, *Intellij*, *Eclipse* e *NetBeans*.

Neste projeto será desenvolvido uma extensão/plugin para o editor de código *Visual Studio Code* capaz de controlá-lo com base em comandos recebidos de uma aplicação externa.

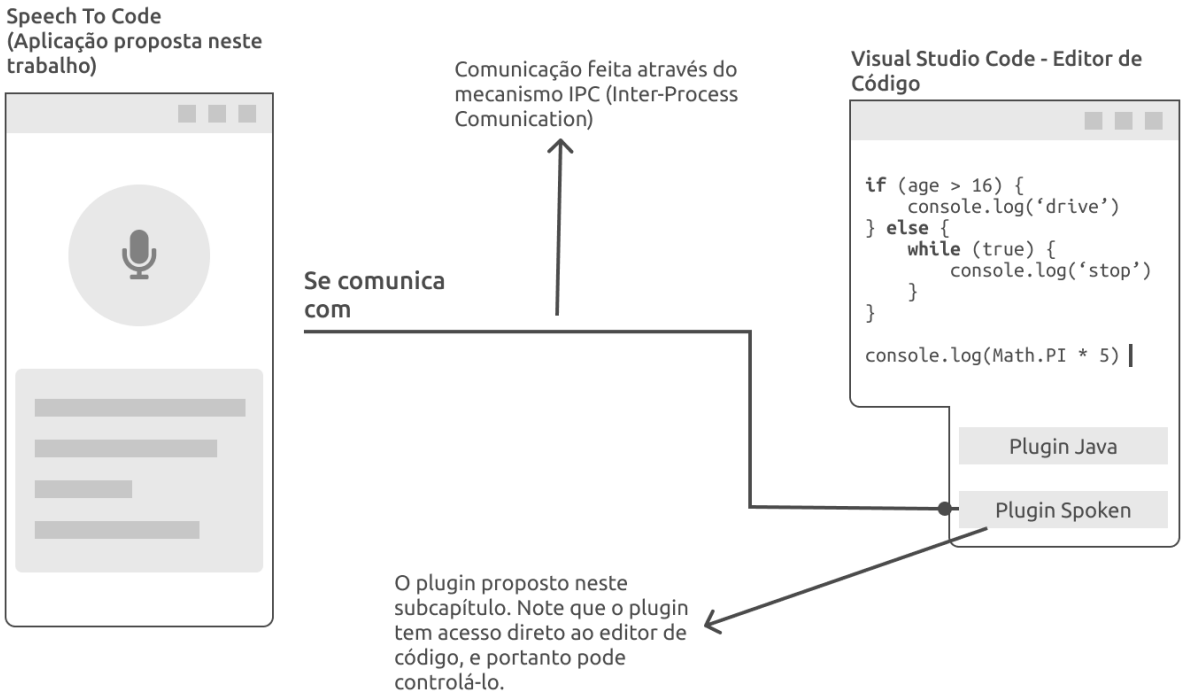

Figura 17. Diagrama mostrando como um plugin do Visual Studio Code pode se comunicar com uma aplicação externa. Fonte: O autor

2.3.2.1 Extensão Spoken Para o Visual Studio Code

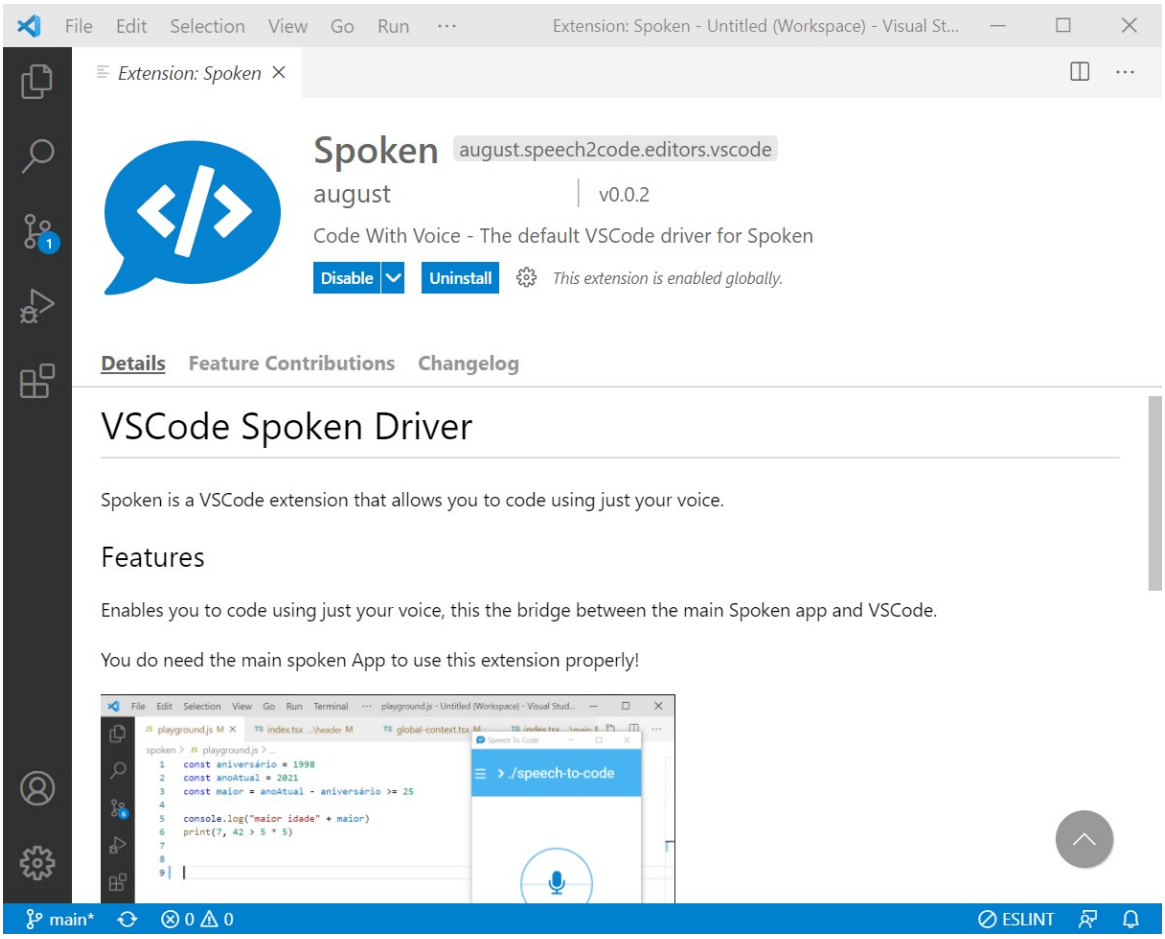

Figura 18. Spoken: A extensão para VSCode proposta e desenvolvida neste projeto. Fonte: *print screen* da extensão Spoken

Spoken é o nome da extensão proposta e desenvolvida neste projeto para o editor de código Visual Studio Code (VSCode). A proposta dessa extensão é agir como um intermediário entre um programa externo que deseja realizar operações no Visual Studio Code e o próprio Visual Studio Code. Como mostrado na figura 17, essa extensão é capaz de receber requisições de um programa externo para a manipulação do editor de código e manipulá-lo como especificado na requisição.

Por se tratar de uma extensão ela roda no mesmo contexto do VSCode e tem acesso a APIs que permitem controlá-lo. Usando essas APIs, que só as extensões têm acesso, é possível controlar (sem a interação do usuário) o VSCode para realizar tarefas como: trocar de linha, escrever texto em qualquer linha, identar o código, executar o arquivo atual, dentre muitas outras.

Essa extensão obedece a um contrato que lista quais interações ela deve ser obrigatoriamente capaz de realizar com o editor de código. Essas interações podem

ser pensadas como funções e algumas delas delas incluem:

Nome	**Descrição**
write(String text)	Escreve algo no editor de código
goToLine(int line)	Move o ponteiro para linha especificada
select(int from, int to)	Seleciona o texto nas coordenadas especificadas.

Tabela 1. Algumas das interações que a extensão é obrigada a ser capaz de realizar com o editor de código.

Programas externos que desejem interagir com o editor de código também devem conhecer esse contrato, eles devem ser capazes de saber quais funções a extensão, nesse caso Spoken, pode realizar no editor de código.

Dito isso, a ação associada ao comando “execute a função bola na linha 42” pode ser interpretada como a chamada da função *goToLine(42)*, que move o ponteiro do editor para a linha número 42, e *write(“bola()”)*, que escreve o texto “bola()” na linha atual.

Por fim, essa extensão é capaz de escutar e executar requisições vindas de outros programas através de um mecanismo de comunicação entre processos baseado em arquivos. A comunicação entre processos (IPC) é o mecanismo que um sistema operacional provê para que processos compartilhem dados entre si. Aplicações que usam IPC seguem o modelo cliente/servidor, onde o cliente requisita por dados e o servidor responde a essas requisições (Microsoft Docs, 2018). Neste projeto a extensão agirá como servidor e todos os programas externos a fim de manipular o VSCode como clientes.

Logo se uma aplicação qualquer A quiser interagir com o VSCode para, por exemplo, remover uma linha, tudo que ela precisa fazer é enviar uma requisição para a extensão Spoken e a linha será removida. Essa requisição deve conter o nome da função a ser realizada e seus argumentos, como visto na tabela 1.

2.4. JAVASCRIPT ECOSYSTEM

Para a implementação deste projeto foi escolhida a linguagem de programação JavaScript devido a sua facilidade de uso, vasto ecossistema e documentação abrangente. Graças ao vasto ecossistema foi possível utilizar a mesma linguagem de programação em todas as frentes do projeto: seja no mecanismo capaz de reconhecer e analisar comandos de voz, no mecanismo capaz de controlar editores de código ou na interface de usuário da aplicação.

A seguir serão listados os frameworks JavaScript utilizados na criação da interface de usuário da aplicação.

2.4.1. Electron Framework

O *Electron* é um *framework* multiplataforma desenvolvido pelo *GitHub*. Seu objetivo é permitir a construção de aplicações para desktops com a utilização de tecnologias usadas em desenvolvimento para web, como *HTML*, *CSS* e *JavaScript* (Noleto, 2020).

Para isso, ele utiliza o *Chromium*, que é a versão código aberto do navegador *Chrome*. A ideia é que a aplicação desenvolvida seja executada sobre ele como se fosse uma aplicação web. Além disso, o *Electron* também utiliza o *Node.js*, componente que oferece recursos adicionais para acessar funções internas do sistema operacional, como permitir o acesso a diretórios e arquivos (Noleto).

De forma simplificada o *Electron* é um framework que permite que aplicações projetadas para web, para serem acessadas usando um browser, funcionem de forma nativa em desktops. Exemplos de aplicativos que fazem uso do framework *Electron* incluem o cliente do aplicativo de mensagens Whatsapp e o próprio editor de código *Visual Studio Code*.

No contexto deste projeto uma das principais vantagens do uso do framework *Electron* é o acesso simplificado ao microfone do computador. Na web o acesso ao microfone e câmera do usuário é padronizado através da *MediaDevices* API. Com *Electron* é possível utilizar esta mesma API em aplicações desktop.

2.4.2. React Framework

ReactJS é uma biblioteca JavaScript para criação de interfaces de usuário desenvolvida pelo Facebook em 2011 que simplifica o processo de desenvolvimento ao introduzir o conceito de componentes reativos reutilizáveis. Utilizando ReactJS é possível pensar em cada elemento de uma página web como seu próprio componente com um estado interno e que pode ser reutilizado em outras partes do código. Graças a sua performance e simplicidade o *ReactJS* foi considerado o segundo framework mais popular no ano 2020, atrás apenas do *JQuery,* segundo pesquisa do Stackoverflow.

Dito isso, neste projeto será usada a biblioteca *ReactJS* para a criação de todos os componentes da interface do usuário.

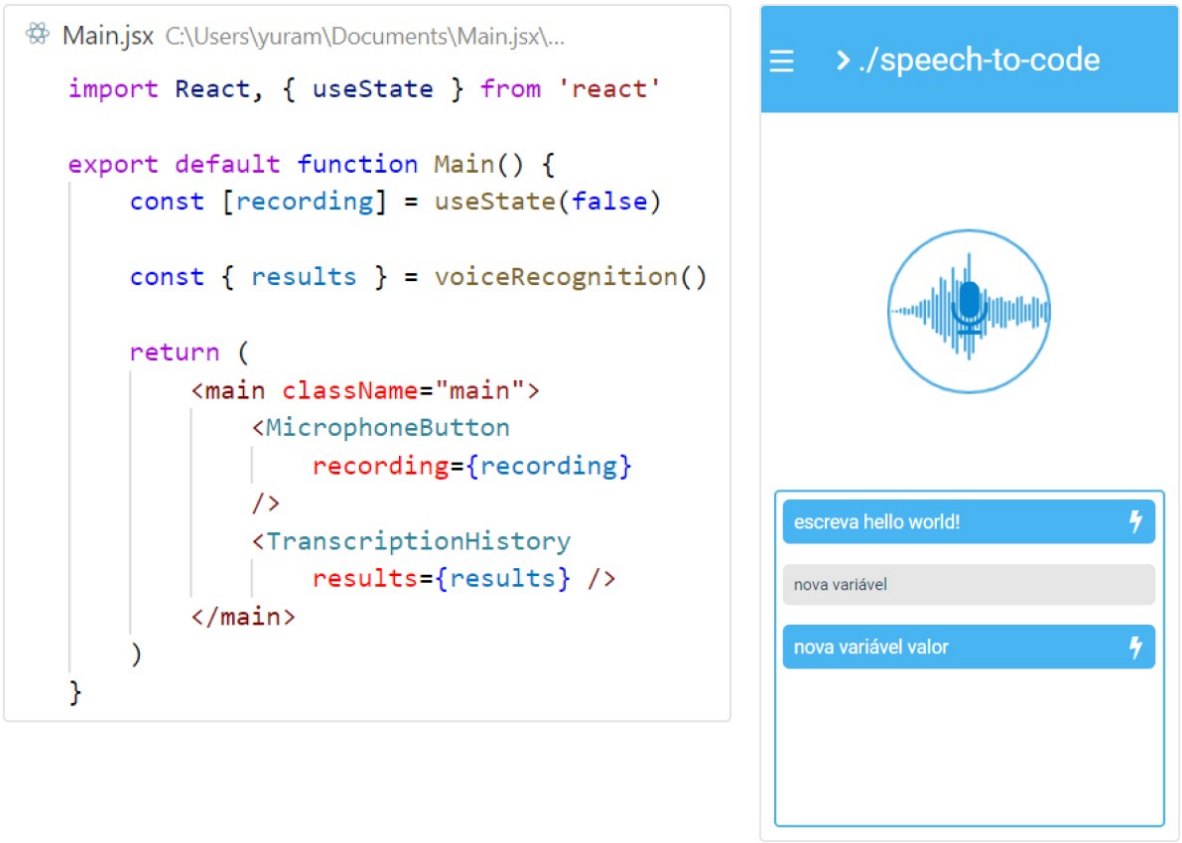

Figura 19. A esquerda o código em ReactJS usado para criar a tela inicial da aplicação, a direita o resultado. Fonte: O autor

2.4.3. Express Framework

Express.js, ou Express, foi o framework utilizado para o desenvolvimento do backend da aplicação. O *Express* é um dos frameworks mais populares no desenvolvimento de servidores graças a sua simplicidade de uso e seu vasto ecossistema de módulos que adicionam funcionalidades pontuais ao framework.

O *Express* é bastante minimalista, no entanto, é possível criar pacotes de middleware específicos com o objetivo de resolver problemas específicos que surgem no desenvolvimento de uma aplicação. Há bibliotecas para trabalhar com cookies, sessões, login de usuários, parâmetros de URL, dados em requisições POST, cabeçalho de segurança e etc (MDN Web Docs, 2020).

Neste projeto o backend da aplicação, implementado com o framework *Express*, será responsável por servir os arquivos estáticos gerados pelo *ReactJS* e implementar um serviço de gerenciamento de tokens para o uso do serviço *Azure Speech To Text*.

3. DESENVOLVIMENTO DA APLICAÇÃO SPEECH2CODE

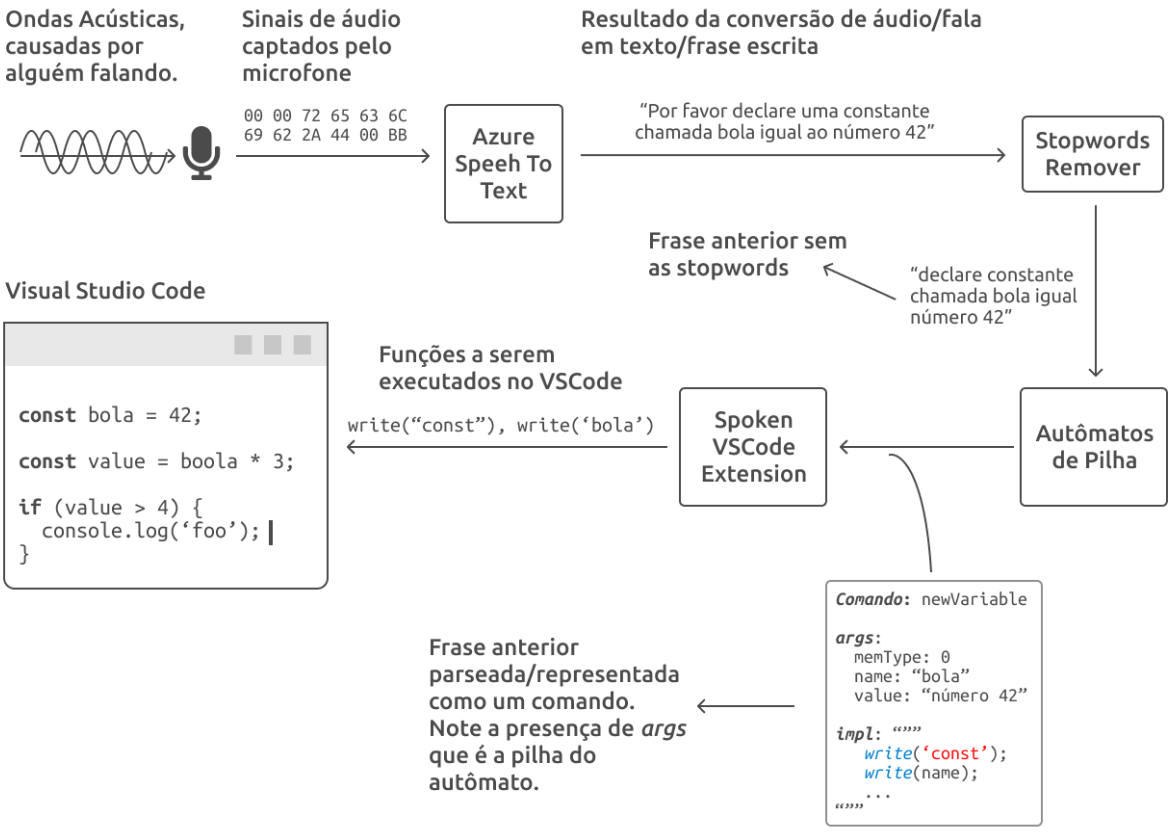

Figura 20. Blueprint da aplicação mostrando como todos os componentes interagem entre si. Fonte: O autor

A seguir serão mostrados como os conceitos e tecnologias descritos no capítulo anterior interagem entre si para alcançar os objetivos propostos deste trabalho.

O processo de desenvolvimento será divido em 3 fases: definição, reconhecimento e análise de comandos de voz, *Spoken VSCode Extension* e a criação da interface de usuário da aplicação.

3.1. COMANDOS DE VOZ

Como visto anteriormente, comandos de voz são frases que o usuário pode dizer em voz alta que resultam em uma ação no editor de código. A seguir será apresentada a definição e como criar um comando de voz e a lista de comandos implementados neste projeto.

3.1.1. DEFINIÇÃO

Um comando qualquer C pode ser encarado como um par (A, P) onde:

- A é uma ação qualquer que deve ser executada quando este comando for accionado.
- P é um conjunto de autômatos capazes de reconhecer frases associadas a este comando.

Logo, se uma frase qualquer x for reconhecida por um dos autômatos de P a ação A deve ser acionada.

Fisicamente um comando pode ser interpretado como uma pasta contendo dois tipos de arquivos:

- Um único arquivo JavaScript chamado "*impl.js"* contendo instruções para a realização do comando.
- Um ou mais arquivos contendo a representação textual dos autômatos de pilha usados para reconhecer frases associadas a este comando. Esses autômatos são representados usando a linguagem de representação de grafos *DOT*.

Novamente, as instruções do arquivo "*impl.js*" só serão executadas para uma frase caso essa seja reconhecida por um dos autômatos que são representados pelos arquivos *DOT*.

Figura 21. Pasta mostrando a estrutura de um comando. Fonte: O autor

Na figura 20 que mostra a definição do comando “estrutura condicional” é possível observar os dois tipos de arquivos mencionados anteriormente. Um chamado *“impl.js”* que representa a ação a ser executada por este comando e outros dois chamados *“phrase_en-US.dot”* e *“phrase_pt-BR.dot”* que representam os autômatos responsáveis por reconhecer frases associadas a este comando.

A seguir será discutido mais a fundo os arquivos “*impl.js*” e “*phrase_pt-BR.dot*” (arquivo DOT).

3.1.1.1. Arquivos DOT

```
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
}
```

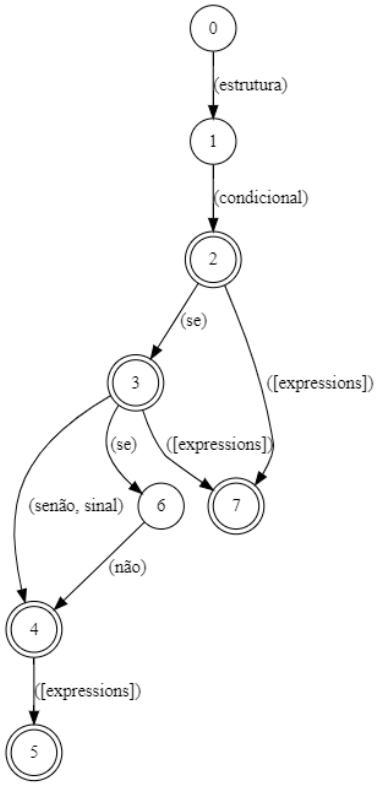

Figura 22. Conteúdo do arquivo “phrase_pt-BR.dot”. A esquerda sua representação textual e a direita visual. Fonte: O autor

Os arquivos “*phrase_en-US.dot*” e “*phrase_pt-BR.dot*” são a representação textual de um autômato de pilha capaz de reconhecer frases relacionadas ao comando “estrutura condicional”. Esses arquivos foram criados usando a linguagem de representação de grafos chamada DOT.

O autômato representado pelo arquivo “*phrase_en-US.dot*” só reconhece frases em inglês enquanto que o “*phrase_pt-BR.dot*” só reconhece frases em português. Caso houvesse a necessidade de suporte a um terceiro idioma, digamos francês, ele poderia ser facilmente adicionado com a adição de um terceiro autômato para reconhecer frases em francês chamado “*phrase_fr-FR.dot*”. A ideia é que não importa se a frase de entrada for “*se cinco é maior que nove então*” (português) ou “*if five is greater than nine then*” (inglês) ou ainda se “*si cinq est supérieur à neuf alors*” (francês) a ação a ser executa é sempre a mesma: aquela associada ao comando “estrutura condicional”. Atualmente a aplicação dá suporte aos idiomas inglês e português.

O autômato de pilha da figura 22 é capaz de reconhecer 8 frases diferentes relacionadas a criar uma estrutura condicional, algumas delas incluem:

Frase	**Pilha**
estrutura condicional	(Ø)
estrutura condicional se senão	(senão)
estrutura condicional *[expression]**	([expression]*)
estrutura condicional se senão *[expression]**	(senão, [expression])*

Tabela 2. Algumas das frases reconhecidas pelo autômato da figura 21.

(*) Qualquer frase reconhecida por um autômato chamado *expresion*.

Qualquer uma dessas frases é capaz de acionar a ação associada ao comando “estrutura condicional”. Como vimos anteriormente a implementação desta ação se encontra no arquivo “*impl.js*”.

3.1.1.2. Arquivo de Implementação

```
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
```

Figura 23. Conteúdo do arquivo “impl.js”. Fonte: O autor

Um arquivo JavaScript contendo apenas uma função que representa a ação

associada ao comando de voz, essa função é executada sempre que um dos autômatos associados reconhecem uma frase como pertencente a este comando.

Esta função tem como objetivo realizar o que foi pedido no comando de voz, para isso ela tem acesso a pilha do autômato que a chamou e a capacidade de interagir com o *VSCode* usando o protocolo descrito na tabela 1. Com a capacidade de interagir com o *VSCode* essa função pode fazer coisas como escrever na tela e trocar de linha e com acesso a pilha do autômato ela sabe *o que escrever na tela* e *pra que linha ir*.

Voltando ao autômato da figura 21 e com a definição de ação apresentada neste subcapítulo podemos completar a tabela 2 da seguinte forma:

Frase	**Pilha**	**Ação**
estrutura condicional	(Ø)	Escreve na tela: ***if (anything) {*** ***}***
estrutura condicional se senão	(senão)	Escreve na tela: ***if (anything) {*** ***} else {*** ***}***
estrutura condicional *[expression]**	([expression]*)	Escreve na tela: ***if ([expression]) {*** ***}***
estrutura condicional se senão *[expression]**	(senão, [expression])*	Escreve na tela: ***if ([expression]) {*** ***} else {*** ***}***

Tabela 3. Algumas das frases reconhecidas pelo autômato da figura 21 e suas ações no VSCode.

(*) Qualquer frase reconhecida por um autômato chamado *expresion*.

Com isso podemos concluir que a ação tomada por um comando depende de qual frase foi usada para acionar tal comando, em outras palavras dada uma frase e uma ação, a ação realizada dependerá do conteúdo da pilha do autômato que reconheceu tal frase.

3.1.2. Lista de Comandos Implementados

No subcapítulo anterior foi demonstrado que criar um comando é na verdade um processo de criar um autômato para reconhecer frases e uma ação que deve ser executada quando uma frase é reconhecida. A seguir serão listados quais comandos foram implementados usando este método, junto a uma breve descrição e exemplos de frases em portugues e inglês que podem ser usadas para ativar cada comando.

Vale notar que todas as ações são aplicadas ao editor de código *Visual Studio Code* e a linguagem de saída é JavaScript. Logo quando a descrição de um comando diz “declara uma variável” deve ser interpretado como “declara uma variável no *Visual Studio Code* usando a sintaxe da linguagem de programação JavaScript”.

Nome	**Exemplos de frases (pt-BR en-US)**	**Descrição**
variableAssign	“nova constante b igual a número 7”, “new constant called graph”, “constant b equals string hello string”	Declara uma variável com nome e valor especificados.
variableRef	"referência constante bola", “constant graph”	Faz referência a uma variável já declarada.
number	“por favor, número 32”, “number 4646”, “número 99”	Escreve o número inteiro especificado.
string	“string a sua idade é string”, “string what is that string”, “texto apenas um teste texto”	Escreve a string especificada. Tudo aquilo entre a palavra string ou texto é considerado como string.
fnCall	“execute a função soma com os argumentos número 10 e string hello string”, “call function list on variable people”	Chama uma função com nome, argumentos e *caller* especificados.
mathExp	“expressão número 4 mais variável boi menos execute a função lista”,	Cria uma expressão matemática com os

	“expression number 4 minus string hello string”	valores especificados. Suporta as operações: adição, subtração, divisão e multiplicação.
logicExp	“expressão execute a função * calcular imc * maior que o número 10”, “expression variable b and variable c”	Cria uma expressão lógica com os valores especificados. Suporta: e, ou, >, <, >=, <=.
condition	“estrutura condicional se expressão número 5 maior que variável bola”, “conditional statement if else expression call function value or variable ball”	Cria uma estrutura condicional se..então com a condição especificada.
goToLine	“vá para a linha 4, por favor”, “linha 10”, “line number 4”,	Troca a linha atual para a linha especificada.
newLine	“linha nova”, “nova linha”, “new line, please”	Adiciona mais uma linha.
select	“selecione da linha 3 até a linha 10”, “select word value”, “selecione da letra e até a letra j”	Seleciona o intervalo entre linhas, letras e palavras especificadas.
run	“execute este arquivo por favor”, “please run this file”	Executa o código do arquivo atual.
newFn	“nova função bola com 3 argumentos”, “new function sum”, “nova função oi retornando número 4”	Cria uma função com nome, valor de retorno e número de argumentos.
repetition	“estrutura de repetição”, “estrutura de repetição do número 4 até o número 43” “estrutura de repetição para todo item em lista”	Cria um laço de repetição com os argumentos providos.
write	“por favor escreva quantos anos você tem”, “please write it down who are you”	Escreve qualquer coisa no editor de código. Tudo aquilo depois da

	“can you please print hello doctor”	palavra “escreva” será escrito.

Tabela 4. Comandos implementados no decorrer deste projeto.

A lista completa desses comandos junto a uma descrição mais profunda, suas implementações e definição dos autômatos de pilha podem ser encontrados em: *https://github.com/pedrooaugusto/speech-to-code/tree/main/spoken/src/modules/typescript#typescript-voice-commands*

3.1.3. Transformando Comandos de Voz em Texto

Até agora, comandos de voz foram tratados como frases em formato textual, neste subcapítulo veremos como transformar frases ditas em voz alta para texto, para que elas possam então ser reconhecidas pelo mecanismo de autômatos já mencionados.

Para transformar frases ditas pelo usuário em texto será usado um produto da Microsoft chamado *Azure Text To Speech*, esse produto te dá acesso ao serviço de conversão de fala em texto da Microsoft, o mesmo usado em sua assistente de voz *Cortana* (Azure, 2020). Por se tratar de um *Web Service* ele pode ser usado em qualquer dispositivo com acesso a internet e funciona no modelo cliente/servidor sobre *websockets*.

Neste projeto será usada uma funcionalidade do *Azure Speech To Text* capaz de fazer reconhecimento de voz em tempo real no áudio captado pelo microfone do dispositivo. A cada segundo o cliente envia um pacote contendo 1 segundo de áudio captado pelo microfone para o serviço de reconhecimento de voz que, com base em todos pacotes de áudio já recebidos, tenta identificar o que foi dito, e então envia de volta ao cliente o seu melhor *palpite* do que foi dito em forma textual.

No contexto desse projeto após receber esse palpite do foi dito em forma textual será usado todo o mecanismo de autômatos já descrito para testar se o que foi dito também é um comando válido.

3.2. Spoken VSCode Extension

```
1   /**
2    * All plugins that wish to comunicate with Spoken
3    * should implment those methods.
4    */
5
6   interface Robot {
7       // Writes something in the editor
8       write(text: string): void
9
10      // Moves the cursor to a different line
11      goToLine(number: string, cursorPosition: 'END' | 'BEGIN'): void
12
13      // ...
```

Figura 24. Exemplo de alguns métodos que a extensão deve implementar por contrato.
Fonte: O autor

O processo de desenvolvimento da extensão *Spoken* consiste em implementar todos os métodos que ela deve dar suporte como estabelecido por contrato. Ao seguir este contrato é garantido que qualquer comando da tabela 3 pode ser executado com êxito por esta extensão. Um exemplo deste contrato pode ser visto na figura 23 que diz que a extensão *Spoken* deve ser capaz de escrever algo no editor código. O próximo passo é fazer exatamente isto utilizando a API que o VSCode expõe para as suas extensões como mostrado na figura abaixo.

```
16      /**
17       * Writes something in the current text input
18       * @param text The text to be written
19       */
20      write = (text: string) => new Promise<void | Error>((res, rej) => {
21          Log('[vscode-driver.robot-vscode.write]: Executing write(' + text + ')')
22
23          const [editor, e] = this.getEditor()
24
25          editor!.edit((editBuilder) => {
26              editBuilder.replace(editor!.selection, '')
27              editBuilder.insert(editor!.selection.active, text)
28          })
29      })
```

Figura 25. Implementação do método write usando as APIs do VSCode. Fonte: O autor

Outra funcionalidade importante dessa extensão é a capacidade de aceitar requisições vindas de outras aplicações para manipular o VSCode. Isso é alcançado usando o método de comunicação entre processos. Ao iniciar a extensão abre um canal de comunicação chamado "*speechtocodechannel*" que qualquer aplicação pode se conectar e "conversar". É através desse canal que aplicações externas enviam os comandos da tabela 1 para serem executados no contexto do VSCode.

3.3. INTERFACE DE USUÁRIO

Todas as telas da aplicação foram criadas com simplicidade e usabilidade em mente. As telas foram primeiro projetadas na ferramenta de design *Figma* e depois implementadas usando a biblioteca de criação de interface de usuário já mencionada *ReactJS*. A seguir serão mostradas as telas e discutidas as funcionalidades de cada uma.

3.3.1. Tela Inicial

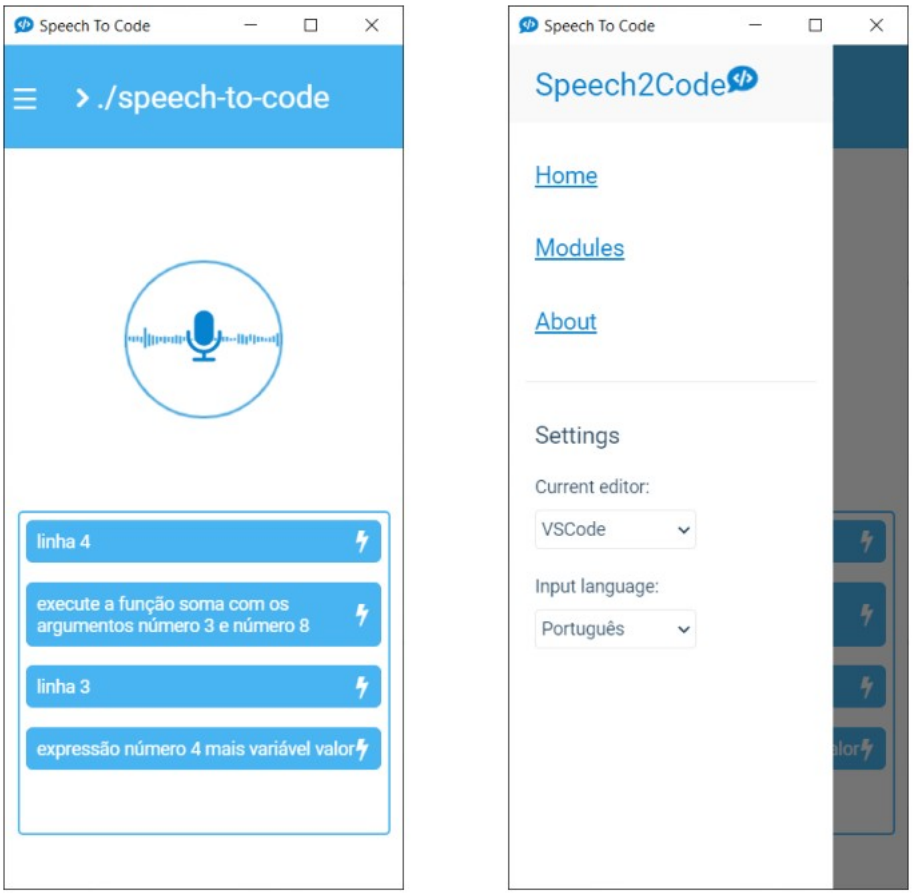

Figura 26. Tela inicial da aplicação. A esquerda em seu estado natural e a direita com o menu lateral expandido. Fonte: print screen da aplicação desenvolvida.

A esquerda na figura acima é possível observar os dois componentes principais que compõem a tela inicial: um círculo com o desenho de um microfone e abaixo uma lista com uma série de frases. O círculo é na verdade um botão que quando clicado começa o processo de reconhecimento de voz que só termina quando o botão é clicado novamente, para denotar que tudo que está sendo dito está sendo gravado e analisado o botão ficará vermelho. Tudo que é dito enquanto o processo de reconhecimento de fala está ativo será transformado em texto e aparecerá na lista de frases, as frases que também forem reconhecidas como comandos aparecerão na cor azul, do contrário cinza. Na figura podemos observar que todas as frases ditas também são comandos válidos, pois estão na cor azul.

À direita da figura 26 podemos ver o menu principal da aplicação que possui 3

ações básicas: alterar o idioma da aplicação entre português e inglês, trocar o editor de código que será controlado e ir para a página de módulos.

3.3.2. Tela de Módulos

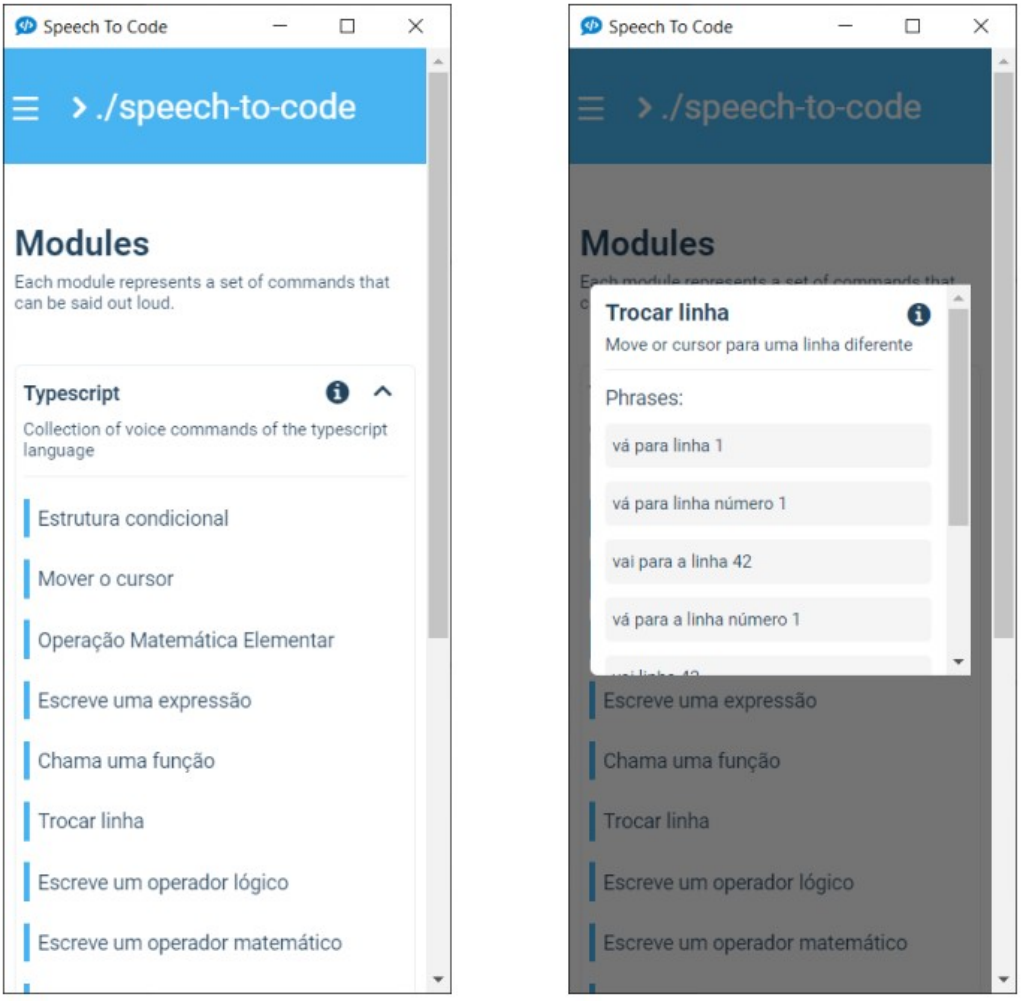

Figura 27. Tela de módulos. Lista todos os comandos da aplicação. A esquerda uma lista com todos os comandos disponíveis, a direita detalhes do comando para trocar de linha. Fonte: *print screen* da aplicação desenvolvida

A tela de módulos lista e mostra detalhes de todos os comandos disponíveis na aplicação e também é sensível ao idioma atual da aplicação, ou seja, se o idioma da aplicação estiver em inglês somente os comandos disponíveis em inglês serão listados. Através dessa tela também é possível seguir um link para mais informações de cada comando.

4. RESULTADOS

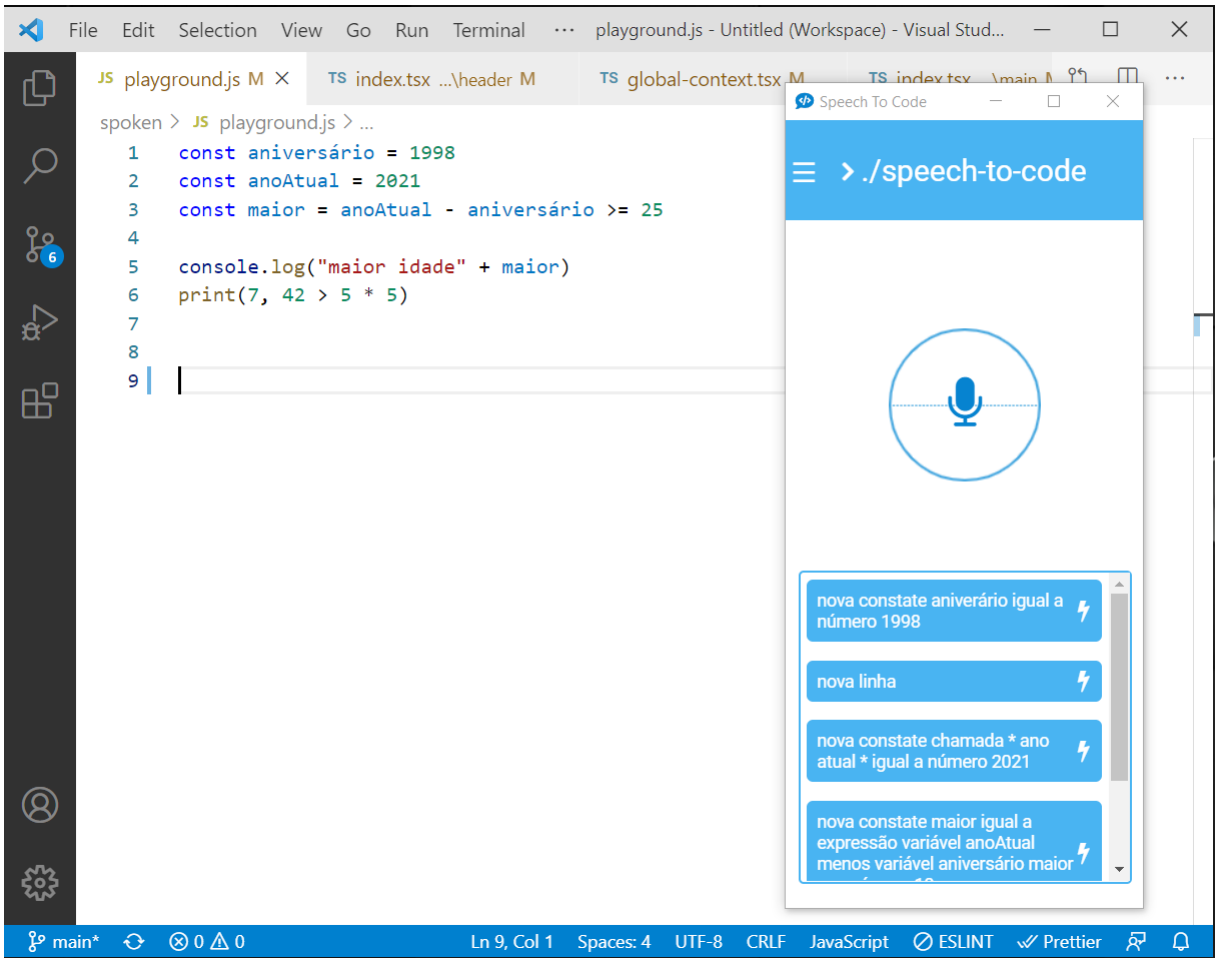

Figura 28. Exemplo da aplicação em funcionamento. A esquerda é possível ver o código gerado e a direita os comandos de voz usados. Fonte: *print screen* da aplicação desenvolvida interagindo com o VSCode.

A proposta inicial deste trabalho era criar uma ferramenta capaz de habilitar programadores a programar em JavaScript fazendo uso da voz. Para isso foram desenvolvidos: uma linguagem livre de contexto representando os comandos de voz disponíveis (tabela 3), uma aplicação capaz de identificar comandos de voz pertencentes a esta linguagem (figura 26) e uma extensão capaz de controlar o editor de código *Visual Studio Code* para executar tal comando (figura 18). O resultado de todos esses componentes trabalhando em conjunto pode ser observado na figura acima que mostra um excerto de código escrito inteiramente utilizando a fala. É possível observar no canto inferior direito que em algum momento a frase "*declare constante chamada aniversário igual a número 1998*" foi dita em voz alta, e identificada como um comando válido. Por isso, a primeira linha do editor de

código mostrado no fundo da imagem é “*const aniversário = 1998*”, exatamente o que foi pedido na frase.

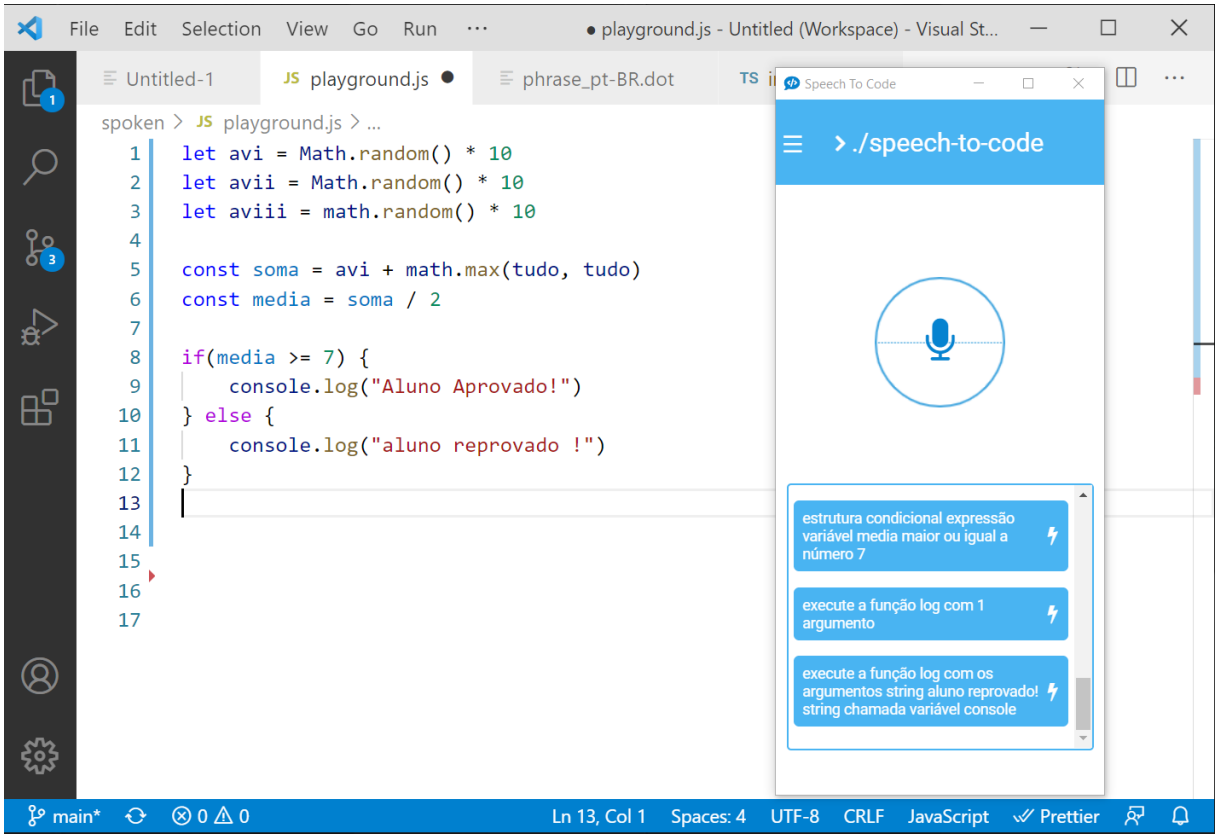

Figura 29. Exemplo da Aplicação em Funcionamento. Fonte: print screen da aplicação

Na figura acima é possível observar um programa para calcular a média entre três notas e mostra na tela “Aprovado”, caso a média seja maior ou igual a 7, ou “Reprovado”, caso a média seja menor que 7. Na criação desse programa foram usadas: estruturas de controle, manipulação de variáveis, chamadas de funções e expressões lógicas e aritméticas. O programa foi criado usando apenas comandos de voz.

Por se tratar de uma linguagem de programação de propósito geral e uma das mais populares no mundo atualmente, JavaScript tem suporte a um número considerável de funcionalidades, desde *async/await* até *bitwise operations* e funções anônimas.

Devido a este grande número de funcionalidades e o tempo limitado para realização deste projeto, foi estipulado um conjunto básico de funcionalidades ao qual a aplicação deve suportar. Isto é, embora a linguagem JavaScript dê suporte a

um número considerável de funcionalidades diferentes, utilizando esta aplicação só será possível utilizar um conjunto limitado dessas funcionalidades. O número de funcionalidades suportadas está diretamente relacionado à tabela 3 e crescerá ao mesmo passo que esta.

A seguir será mostrado os conjuntos de funcionalidades básicas estipulados e se eles foram implementados ou não junto a um exemplo na linguagem JavaScript.

Conjunto	**Descrição**	**Exemplo**	**Implementado**

Manipulação de variável	Declarar e fazer referência a variáveis já declaradas.	const a = 7;
		const b = 6;
		const c = a * b;
			SIM.

Estruturas Condicionais	Criar estruturas condicionais.	if (3 > 4) {} else {} switch(a) {...}	PARCIALMENTE. Suporta: if...else Não suporta: *shor-circuit* e *switch*
Funções	Criar e executar uma função com argumentos.	Math.max(0, 1); hello(); test(“oi”);	SIM.
Estruturas de repetição	Criar estruturas de repetição.	for(...) {} while(...) {}	SIM.
Tipos Primitivos	Manipulação de tipos primitivos da linguagem.	let a = “hello” let b = true let c = 42	PARCIALMENTE. Suporta: números inteiros e strings. Não suporta: null, undefined e booleans.
Operações lógicas e matemáticas.	Operações lógicas e matemáticas que manipulam 2 ou mais termos.	42 * 7 > 34 ~(a && b) 1 + 2 + 3 / 5	SIM.
Tipos complexos	Permite a criação e manipulação de tipos não primitivo (classes/structs).	new Date(42) class Hello {...} {name: “pedro”}	NÃO.

Tabela 5. Conjunto de funcionalidades básicas estipuladas.

4.1. Requisitos para execução da aplicação

Os requisitos mínimos para o pleno funcionamento da aplicação são os seguintes:

- **Rede:**
  Conexão estável e veloz com a internet;
- **Sistema Operacional:**
  Testado apenas no Windows 10;
- **Memória:**
  Pelo menos 3GB de ram e 270MB de espaço em disco disponível;
- **Python:**
  Testado com a versão 3.9.1 do Python;
- **Visual Studio Code:**
  Testado com a versão 1.55.2 do VSCode;

5. CONCLUSÃO

Neste trabalho foi proposta e desenvolvida uma aplicação de tecnologia assistiva capaz de habilitar programadores a programar em JavaScript sem o uso das mãos, usando apenas comandos de voz. O intuito da aplicação é habilitar programadores incapazes ou com grande dificuldade de fazer uso das mãos, seja por lesão ou deficiência, a continuar exercendo sua profissão. Seu desenvolvimento foi baseado nos conceitos de teoria dos autômatos, reconhecimento de fala e automação de interface do usuário. Esta mesma metodologia pode ser empregada na criação de outras aplicações com propósito similar, como por exemplo, uma ferramenta que além de JavaScript também suporta Python ou que seja capaz de manipular o terminal bash do linux.

Além de aumentar o número de funcionalidades da linguagem JavaScript suportadas, efetivamente completando as tabelas 3 e 4, trabalhos futuros incluem algumas melhorias no reconhecimento e análise de comandos de voz:

- Treinar e criar um modelo de reconhecimento de fala customizado no *Azure*

Speech To Text só com as frases que representam comandos. Isso fará com que o reconhecedor de fala dê preferência a estas frases. Um exemplo disso é a frase usada para escrever algo na tela: "*write something*". Ao dizer "*write test*" em voz alta muitas vezes isto é reconhecido como "*right test*", pois as palavras "*write*" e "*right*" têm pronúncia similar. Isso pode ser resolvido com a criação de um modelo customizado ou a adição de mais contexto à frase.

- Avaliar a substituição do mecanismo para reconhecer e analisar comandos, que é baseado em autômatos de pilha, por soluções mais robustas baseadas em machine learning como o *Microsoft Azure LUIS* ou o *Google Cloud Natural Language AI*.

REFERÊNCIAS BIBLIOGRÁFICAS

PUPO, Deise Tallarico; MELO, Amanda Meincke; PÉREZ FERRÉS, Sofia. **Acessibilidade: Discurso e Prática no Cotidiano das Bibliotecas**. São Paulo, SP: UNICAMP, 2008.

BERSCH, Rita. O que é Tecnologia Assistiva ?. **Tecnologia assistiva**. Disponível em: <https://www.assistiva.com.br/tassistiva.html>. Acesso em: 05 maio. 2021.

Assistive Technology Australia. What is Assistive Technology ?. **Assistive Technology Australia**. Disponível em: <https://at-aust.org/home/assistive_technology/assistive_technology>. Acesso em: 05 maio. 2021.

PREVIDÊNCIA SOCIAL, Previdência Social. Dados estatísticos - Previdência Social e INSS: **Governo Federal**, 2019. Disponível em: <https://www.gov.br/previdencia/pt-br/acesso-a-informacao/dados-abertos/previdencia-social-regime-geral-inss/dados-abertos-previdencia-social>. Acessado em: 15 de maio de 2021.

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

IBM CLOUD, IBM Cloud Education. Speech Recognition: **IBM Cloud Education**, 2020. Disponível em: <https://www.ibm.com/cloud/learn/speech-recognition>. Acessado em: 15 de maio de 2021.

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

SILVA, Anderson. **Reconhecimento De Voz Para Palavras Isoladas**. Orientador: Ren Tsang. 2009. 60. Monografia – Engenharia da Computação, Universidade Federal de Pernambuco, Pernambuco. 2009. Disponível em: <https://www.cin.ufpe.br/~tg/2009-2/ags.pdf>. Acesso em: 15 de maio de 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. Disponível em: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html>. Acessado em: 15 de maio de 2021.

P. Gouda, Prabhakar. **Application of Finite automata.** 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. Disponível em: <https://www.graphviz.org/pdf/dotguide.pdf>. Acessado em: 15 de maio de 2021.

UIPATH, UIPATH. **What is robotic process automation?.** UI PATH, 2015. Disponível em: <https://www.uipath.com/rpa/robotic-process-automation>. Acessado em: 15 de maio de 2021.

Al Sweigart. **Welcome to PyAutoGUI's documentation!.** PyAutoGUI, 2017. Disponível em: <https://pyautogui.readthedocs.io/en/latest/index.html>. Acessado em: 15 de maio de 2021.

Microsoft Docs. **Interprocess Communications.** Docs Microsoft, 2018. Disponível em: <https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications?redirectedfrom=MSDN>. Acessado em: 15 de maio de 2021.

Cairo Noleto. **Electron: O que é e como criar aplicações utilizando esse framework!.** Trybe, 2020. Disponível em: <https://blog.betrybe.com/framework-de-programacao/electron/>. Acessado em: 15 de maio de 2021.

Stackverflow. **Most Popular Technologies: Frameworks.** Stackoverflow, 2020. Disponível em: <https://insights.stackoverflow.com/survey/2020#most-popular-technologies>. Acessado em: 15 de maio de 2021

MDN Web Docs. **Introdução Express/Node.** Developer Mozilla, 2020. Diponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introduction>. Acessado em: 18 de maio de 2021.

Azure. **Conversão de Fala em Texto.** Microsoft Azure, 2020. Disponível em: <https://azure.microsoft.com/pt-br/services/cognitive-services/speech-to-text/#overview>. Acessado em: 19 de maio de 2021.

Maria Bruna. **Lesão por esforço repetitivo (LER/DORT).** Drauziovarella. Disponível em: <https://drauziovarella.uol.com.br/doencas-e-sintomas/lesao-por-esforco-repetitivo-ler-dort/>. Acessado em: 20 de maio de 20221.