2024

Programming by Voice

Sadia Nowrin
Michigan Technological University, snowrin@mtu.edu

PROGRAMMING BY VOICE

By
Sadia Nowrin

A DISSERTATION
Submitted in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY
In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY
2024

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor:	*Dr. Keith Vertanen*
Committee Member:	*Dr. Laura Brown*
Committee Member:	*Dr. Leo Ureel*
Committee Member:	*Dr. Patricia Ordonez*
Department Chair:	*Dr. Zhenlin Wang*

Contents

List of Figures

List of Tables

Author Contribution Statement

Chapter 2, parts of Chapter 3, and parts of Chapter 4 of this dissertation are based on previously published works, and some passages quoted verbatim from [43], [41] and [42] respectively. Additionally, some sections of Chapters 2 and 3 have appeared in another article [40]. Diagrams and tables are reproduced exactly as they appear in the original publications. The respective author holds the copyright for each of these published works.

Acknowledgments

I would like to express my heartfelt gratitude to Dr. Keith Vertanen. His belief in my potential kept me motivated even during the toughest times.

Special thanks to my committee members, Dr. Leo Ureel and Dr. Laura Brown, for their insightful feedback, genuine interest in my study, and encouragement. I am deeply grateful to Dr. Patricia Ordonez for her invaluable assistance in helping with the study and her encouragement throughout my PhD.

I extend my heartfelt thanks to Steven Ott, Sarah Whittaker, Isabella Gatti, and Alex Gore from Michigan Tech and Dyanlee de Leon Cordero, Christopher Ayala, Andres M. Rosner Ortiz, and Angel J. Ramos from the University of Puerto Rico, Rio Piedras, for their contributions to the voice programming user study.

I also wish to acknowledge the Computer Science Department at Michigan Tech and the Michigan Tech Graduate School for their financial support.

Finally, I am immensely thankful to all the participants in the user studies. This journey has been challenging and rewarding, and I am grateful to everyone who has been a part of it.

Abstract

Programmers typically rely on a keyboard and mouse for input, which poses significant challenges for individuals with motor impairments, limiting their ability to effectively input programs. Voice-based programming offers a promising alternative, enabling a more inclusive and accessible programming environment. Insights from interviews with motor-impaired programmers revealed that memorizing unnatural commands in existing voice-based programming systems led to frustration. In this work, we explore how programmers naturally speak a single line of code and present a comprehensive methodology for a voice programming system aimed at making programming more accessible for diverse users.

To achieve this, we adopted a two-step pipeline. The first step focuses on recognizing single lines of spoken code by adapting a large pre-trained speech recognition model. By adapting the model with just one hour of spoken programs and leveraging existing natural English language data, we reduced the word error rate from 28.4% to 8.7%. Additional improvements were achieved by decoding with a domain-specific N-gram model and rescoring with a fine-tuned large language model tailored to programming languages, resulting in a WER of 5.5%.

The second step involves translating the recognized text into the target line of code. Our approach to text-to-code translation is the first to address spoken programs, converting a single line of text to a single line of code, whereas current systems typically translate comments to blocks of code. We used a large language model known for generating code from comments and adapted it to learn how to generate single lines of code. This adaptation led to a significant improvement in the CodeBLEU score from 56.9% to 83.3% on our test set. In addition, when translating recognized transcripts to target code, our best-adapted model showed marked success. The CodeBLEU score improved from 53.7% to 76.7%, demonstrating the model's ability to handle errors from the speech recognizer.

Chapter 1

Introduction

1.1 Motivation

The traditional reliance on keyboards and mice for programming presents significant challenges, particularly for those with motor impairments. While typing is the standard method for code input, it creates barriers that can prevent individuals with motor impairments from pursuing or advancing in computer science. Furthermore, the physical strain of prolonged typing can result in Repetitive Strain Injury (RSI). RSI can severely limit a person's ability to type, effectively removing them from their profession despite their skills and passion for programming.

Voice programming offers a compelling alternative, allowing code input through speech rather than typing. This approach benefits those with existing motor impairments and serves as a preventive measure against RSI for all programmers. With the help of advancements in speech recognition and natural language pro-

cessing, voice programming has the potential to revolutionize the coding process, making it more inclusive and less dependent on traditional input devices.

1.2 Challenges in Speaking Code

Speaking code is different than typing code. Transitioning from typing to speaking code introduces challenges that must be addressed to develop an effective voice programming system. While programming languages have strict grammar rules that should theoretically make them more predictable than natural language, several factors complicate this predictability:

- **Variation in Spoken Programs:** Different programmers may speak a line of code differently. Novice programmers might struggle with speaking code correctly, while experts might use shorthand or more advanced terminology. A single line of code can be spoken in various ways. For instance, the statement `"System.out.println("Hello, World");"` could be spoken precisely (e.g., "`System dot out dot println open parenthesis quote Hello comma World quote close parenthesis semicolon`") or more naturally (e.g., "`print Hello, World to the console`"). The latter approach is more intuitive and reduces the cognitive load on the programmer, as it does not require the programmer to recall the exact syntax. However, this natural method can lead to misinterpretation. For example, the command `"Create a for loop that iterates from 1 to 10"` neither specifies whether the loop counter should be declared within the loop or separately nor details how it should

increment the variable and the name of the variable.

- **User-defined Symbols:** Dictating user-defined symbols such as variable names, method names, and class names can be particularly challenging. For example, a variable named "`userAge`" might be spoken as "`user age`" or "`usersage`", or with capitalization specified as "`user uppercase a age`". Furthermore, abbreviations and numbers mixed with words can add to the complexity. For instance, a method named "`get3rdItem`" could be spoken as "`get third item`" or "`get three rd item`". A particularly ambiguous case might involve a variable name that can be spelled out or spoken as the full name, like "`cnt`", which could be spoken as "`c n t`" or "`count`".

- **Mixed Natural Language and Programming Terms:** Another significant challenge arises from the mix of natural language and programming-specific terms in comments and documentation. For instance, a user might say, "`set user status to null if no record is found`", where the programming term "`null`" is mixed with natural language. Another example could be "`comment: validate emailAddr before submitting`", where an abbreviated variable name is interspersed with natural language. Additionally, a user might say, "`check if isActive boolean is true before processing`", combining the programming term "`boolean`" with natural language.

1.3 Current Landscape of Voice Programming

Developing a voice programming system is complex, as it involves more than merely dictating lines of code. It requires accurately recognizing domain-specific terminology, handling diverse accents, integrating with existing programming environments, and managing complex programming constructs and syntax. This chapter presents the overall picture of the current landscape of voice programming systems. Details about user studies, methodology, and the evaluation of the systems will be provided in Chapter 3.

Several software developers with motor impairments have developed voice programming systems for personal use and for others willing to invest significant effort in learning a new spoken language. Tavis Rudd, a software developer who suffered from RSI, created a voice-based system to write code. His spoken Python language [54] requires users to learn over 1000 specific commands. For example, users must say "`snake case underscore variable`" to input a variable name like "`snake_case_variable`". Tavis Rudd's system uses the Dragon Naturally Speaking recognizer, customized extensively to handle the programming-specific vocabulary. Dragon Naturally Speaking [58] is a speech recognition engine designed to dictate natural text. In a study [20] evaluating Dragon Naturally Speaking, it achieved a WER of 3-5% for Python, but 13-18% for C and 20-27% for Java. Its use in programming is thus limited by its general-purpose nature, requiring significant customization and scripting to support programming languages effectively.

After developing severe hand pain, Ryan Hileman left his full-time software

engineer job to develop Talon[1]. Talon allows hands-free input through speech recognition and eye tracking. Rick Mohr, also diagnosed with RSI, developed Vocola[2], a spoken command language to control a computer. Additionally, Matt Wiethoff, after his RSI diagnosis, developed Serenade[3], a popular voice programming platform. Serenade requires precise spoken commands. Talon, Vocola, Serenade, and Tavis Rudd's developed system all require commands to match predefined sets exactly, lacking natural language processing capabilities. The limitation here is the steep learning curve due to the vast number of commands.

Previous research has predominantly focused on command-based voice programming systems, which necessitate that programmers learn specific grammatical structures and syntax to function effectively [35, 51, 37, 66, 22]. These systems typically involve users issuing predefined commands that the system recognizes and converts into code. Some studies have investigated the potential for users to create custom commands tailored to their individual needs [35]. Furthermore, there have been efforts to integrate voice programming into existing development environments, which leverages the familiar tools and workflows that programmers already use [7, 37, 35].

Evaluations of command-based systems [35, 51, 37, 66, 22] with motor-impaired programmers have been limited, although some studies have aimed to address this gap [51, 37]. The existing command-based systems have often highlighted challenges such as limited speech recognition accuracy and inefficiency in real-

[1]https://talonvoice.com/
[2]http://vocola.net/
[3]https://serenade.ai/

world scenarios. The limited accuracy of speech recognition systems has been a significant barrier, often resulting in incorrect code generation and increased user frustration. Additionally, the inefficiency of these systems in real-world programming environments, where quick and accurate input is required, has further limited their adoption. These challenges underscore the need for improved speech recognition technology that can more accurately interpret spoken commands and better support the needs of all programmers.

In addition to command-based systems, block-based voice programming tools [66, 44, 27] have been developed, allowing users to construct programs using a more visual interface. However, they still suffer from the fundamental issue of speech recognition limitations. According to their evaluation, the system was easy to use, but users found it hard to learn the predefined commands. The authors concluded that more accurate speech recognition is required as short commands like `"in"` or `"up"` were misrecognized 70% of the time. While these visual programming languages are primarily designed for educational purposes and are rarely used by professional programmers, they play a crucial role in introducing programming concepts to novices.

Researchers have explored developing natural language-based systems that enable users to speak more naturally without needing to learn specific commands or structures [3, 6, 15]. Some have even developed conversational systems [63] to facilitate programming through natural language interactions. CONVO supports program creation, editing, and execution using voice and text inputs. The system allows users to issue commands such as "create a variable" and engage in a dialogue

to complete programming tasks. While CONVO demonstrates the potential of conversational programming, it faces significant challenges, particularly in speech recognition accuracy. Using Google's Cloud Speech-To-Text API, the study found that current ASR systems often misinterpret programming-specific commands and highlighted the need for custom ASR models tailored to programming languages. However, these systems are still underdeveloped: they struggle with accurately recognizing and interpreting domain-specific terminology.

Despite progress in voice programming, challenges like speech recognition accuracy and needing precise commands are still major barriers. Current systems often require users to remember many voice commands or need a lot of customization. While customization can reduce the need to remember specific commands for simple programming statements, it comes with challenges. Customization can be time-consuming and may not scale well as programming tasks become more complex. As the number of required commands grows, managing these customizations can become cumbersome, highlighting the limitations of current voice programming systems. This underscores the need for improved solutions that can intuitively understand and execute natural language instructions with minimal user training or customization.

1.4 Our Approach

Developing a voice-based programming system is a challenging endeavor that requires sophisticated speech recognition models and intelligent translation mech-

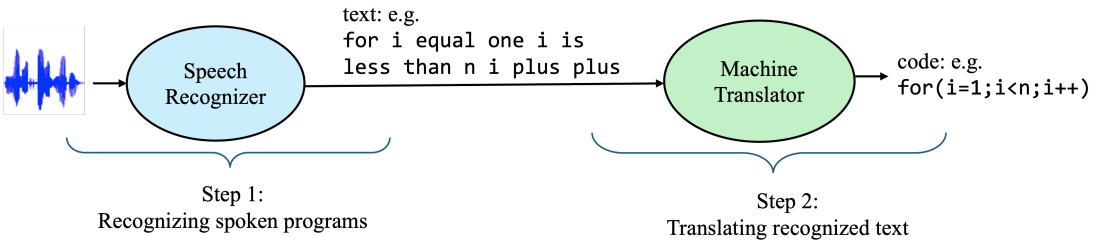


Figure 1.1: A two-step pipeline of a voice programming system.

anisms to handle the variability and complexity of spoken code. Previous approaches have relied on unmodified speech recognizers trained in natural English. These systems often fail to accurately capture the nuances and specific syntax of programming languages, leading to frequent misinterpretations and errors in code generation. Our research represents the first work to build a voice programming system from the ground up, collecting training data specifically tailored for recognizing spoken code. This method allows us to develop purpose-built components for the task, significantly improving speech recognition accuracy for programming.

Our research focuses on creating a voice programming system tailored to Java. However, the methods and insights gained from this work are broadly applicable to other programming languages. We aim to develop methodologies that can be adapted for different programming languages, ultimately contributing to the development of a versatile voice programming system.

1.4.1 A Two-Step Pipeline

We adopted a two-step approach: first, recognizing spoken code, and second, translating the recognized text into accurate code as shown in Figure 1.1. The

two-step pipeline offers several key benefits: by breaking down the task into distinct steps, we can address the challenges of speech recognition and code generation separately, leading to a more adaptable system. Additionally, each model can be trained and fine-tuned to handle specific speech types, such as naturally spoken programs or command-based spoken programs, thereby improving recognition accuracy and translation precision. Furthermore, each component can be optimized independently to support different target languages. In contrast, an end-to-end model would require significantly more acoustic data to handle diverse accents and speech variations, making it less efficient and flexible.

The first step focuses on speech recognition, converting spoken input into textual form. This step leverages state-of-the-art commercial and advanced neural research recognizers to capture the diverse ways programmers speak code. The second step involves translating this text into correct code and handling the syntactic and semantic requirements of the target programming language. Separating speech recognition from code generation allows for targeted improvements in each area. It provides a more manageable framework for developing an effective voice programming system by focusing on the unique challenges each step presents.

1.4.2 A Line-by-Line Approach

We focus on a line-by-line approach to voice programming instead of generating whole code blocks. Several key reasons support our choice of the line-by-line approach:

- **Accuracy and Trust Issues:** Large language models (LLMs) [49, 10, 12,

29] have revolutionized programming by generating code blocks from text prompts. However, their accuracy can vary significantly, largely depending on the quality and scope of their training data [61]. According to the 2023 StackOverflow survey[4], 70% of the 89,184 programmers surveyed were either using or planning to use AI tools in their development process that year, with 32% noting that these tools have significantly increased their productivity. Despite this, only 2.85% of respondents highly trusted the output, and 39.3% somewhat trusted it. When LLMs struggle to produce accurate results, it often becomes necessary to correct individual lines or sections of the generated code manually.

- **Non-Linear Workflows:** Programmers often work in non-linear fashions, jumping between code sections or revisiting previously written lines. They may prefer incrementally breaking down tasks into smaller, manageable pieces and writing code.

- **Learning and Skill Retention:** LLMs can undoubtedly automate routine and repetitive coding tasks, freeing up programmers' time for more complex and creative aspects of development. However, an over-reliance on LLMs to generate entire code blocks could hinder novices' understanding of fundamental coding concepts and problem-solving skills. Even for experienced developers, excessive dependence on LLMs could diminish their manual coding skills and ability to debug effectively, potentially impacting

[4]https://survey.stackoverflow.co/2023/

their long-term career prospects.

- **Control and Precision:** Speaking code line-by-line gives programmers more control and accuracy, ensuring the final code meets their needs.

Based on these insights, we collected data on how programmers speak individual lines of code rather than entire programs. We adapted LLMs originally designed to handle whole code blocks to recognize and process code spoken line-by-line better, aiming to improve the accuracy of voice-driven coding and ensure programmers can maintain control over the coding process.

1.5 Research Questions

Our work addresses the following research questions:

1. RQ1: What are motor-impaired programmers' perceptions, needs, and challenges regarding voice programming systems?

2. RQ2: How do different programmers naturally speak programs without prescriptive grammar?

3. RQ3: How can we improve the accuracy of recognizing line-by-line spoken programs?

4. RQ4: How can we accurately translate the recognized text into the target line of code?

1.6 Dissertation Outline and Contributions

The organization of the rest of the dissertation is as follows:

- **Chapter 2:** This chapter presents an interview study with motor-impaired programmers. The contributions include:
  - Providing a comprehensive understanding of the unique challenges faced by motor-impaired programmers, offering valuable insights for designing more accessible and effective voice programming systems.
  - Highlighting motor-impaired programmers' needs and preferences, which can guide the development of tailored solutions to enhance their programming experience.
- **Chapter 3:** This chapter investigates how novice and expert programmers naturally speak a line of code. The contributions include:
  - Providing new insights into how diverse programmers vocalize code without being taught specific grammar. This can inform the development of more effective voice-based systems.
  - Conducting the first comparison of two methods for collecting spoken code: speaking a missing and highlighted line of code. This helps identify the more effective approach for capturing necessary variations and developing accurate language models.
  - Releasing the first dataset[5] that includes individual spoken lines of

[5] https://osf.io/h6nk4

code, corresponding target lines of code, and transcripts of how programmers vocalized those lines, offering a valuable resource for the research community.

- **Chapter 4:** This chapter focuses on the first step of our two-step pipeline: recognizing the literal words spoken by programmers. The contributions include:
  - Conducting the first study that recognizes line-by-line spoken programs by adapting a speech recognizer with knowledge of natural language using our collected spoken programs.
  - Improving line-by-line spoken program recognition by adapting a large pre-trained language model trained on various programming languages.
- **Chapter 5:** This chapter focuses on the second step: converting recognized text into the target line of code. The contributions include:
  - Demonstrating methods to adapt a large language model for incremental code generation originally trained to generate whole code blocks.
  - Illustrating how considering the context surrounding each line of code can significantly improve the model's ability to produce syntactically and semantically correct code.
- **Chapter 6:** This chapter concludes the dissertation by summarizing the contributions, discussing limitations, and suggesting future work.

Chapter 2

Interviewing Motor-impaired Programmers

2.1 Introduction

The first step in developing a voice programming system that effectively meets user needs is to thoroughly understand the challenges and requirements of the target users. For programmers with motor impairments, programming by voice can significantly enhance their productivity by providing an alternative to traditional input methods. This chapter explores the experiences and perceptions of programmers with motor impairments using voice programming tools. The goal is to uncover their vision for an ideal voice programming system. This insight will guide future development efforts to create a system that truly benefits these users.

2.2 Related Work

A 2018 article [39] described three motor-impaired programmers' experience of using existing voice programming systems to continue their work despite motor impairment. These individuals turned to voice coding to alleviate the strain of traditional typing. The article highlighted the extensive setup and training required to become proficient with voice coding tools. The article also mentioned that learning and adapting to the necessary voice commands often took months of effort. Users reported benefits such as reduced physical strain and maintained productivity. However, they also faced significant challenges due to the lack of natural language processing in existing systems, which required memorizing precise command phrases. Building on this, our interview study of seven motor-impaired programmers provides new insights into some of the significant challenges in programming by voice (e.g. speaking variable names and editing code) and how these programmers envision an ideal voice programming system. To our knowledge, no other studies have interviewed target users in the context of voice programming.

2.3 Participants

We recruited seven motor-impaired programmers with between 6 and 20 years of programming experience. All participants were native English speakers who wrote programs almost daily, utilizing a combination of mouse, keyboard, and voice for programming tasks. They also used voice user interfaces for everyday activities

such as writing emails and getting directions. The participants were recruited through word of mouth. Table 2.1 provides detailed participant information.

2.4 Procedure

We conducted semi-structured interviews with the participants, beginning with obtaining informed consent from each participant. We explained the study's objectives, tasks, and potential risks and emphasized that participation was entirely voluntary. The study comprised a questionnaire followed by a series of open-ended questions.

The questionnaire gathered demographic information, participants' experiences with voice user interfaces, and their programming backgrounds. It also included a 7-point Likert scale to rate their programming experience. Appendix A includes the questionnaire and the open-ended questions.

During the open-ended questions, participants described their use of voice user interfaces for non-programming tasks, shared their experiences with voice programming tools, and discussed their challenges when inputting programs by voice. We also asked them to envision an ideal, highly accurate future voice programming system and identify the most challenging aspects of programming by voice. Additionally, participants were asked about any privacy or social concerns regarding voice programming systems and in what situations they believed voice programming would be most useful. The interviews were conducted via videoconferencing software. Participants were compensated $20 for their participation.

P.	Gender	Current job	Physical condition	Prog. Exp.
P1	female	graduate student	chronic musculoskeletal pain	8 years
P2	male	web developer	neurological accident	7 years
P3	female	graduate student	spinal muscular atrophy	20 years
P4	female	academic researcher	small fiber neuropathy	16 years
P5	male	professor	congenital upper limb deficiency	17 years
P6	male	academic researcher	upper limb musculoskeletal disorder	6 years
P7	male	software architect	temporary repetitive strain injury	24 years

Table 2.1: Details about the interviewed participants with motor impairments. (P. = Participant, Prog. Exp. = Programming Experience)

We transcribed the interview recordings and reviewed the transcripts to understand the participants' perspectives better. Using thematic analysis, we derived codes from the interview data and grouped similar codes into broader themes. Finally, we cross-checked the themes with the original transcripts to ensure they accurately represented the participants' views.

2.5 Results

Our questionnaire revealed six participants used voice interfaces to write emails and get directions. Two participants strongly agreed, and four agreed that speech interfaces sometimes misunderstood them. Five participants strongly agreed that they were expert programmers. Four agreed, and three strongly agreed that they frequently wrote programs.

From the thematic analysis of the qualitative data, we found five themes: experience, vision, challenges, privacy, and usefulness. The resulting themes, short descriptions, and corresponding quotes are presented in Table 2.2. A detailed

discussion of each theme follows, providing a deeper insight into the participants' perspectives and experiences.

Theme: Experience. All participants indicated they took frequent breaks while writing programs. All participants used both Dragon Naturally Speaking[1] and VoiceCode[2]. Three of them currently use Talon[3] to dictate programs. One participant said that existing voice programming technologies can dictate simple words accurately, such as print, insert tabs, and add punctuation, such as single and double quotes. Two participants stated that the technologies they used to program by voice were poorly documented. In addition, the interviewees mentioned that the existing systems forced them to learn a large number of commands. Learning these commands required many hours, and the process was frustrating and stressful. Two participants said that the system occasionally misrecognized a very long command, and they found it annoying and time-consuming to input the command again.

Theme: Vision. Five participants wished they could write programs via natural language. They also mentioned that there should be a mechanism to distinguish between words that sound the same in English but signify different things in the context in which they were stated (e.g. the word `"to"` and the number `"two"`). Participants thought an intelligent voice programming system should be able to incorporate corrections. According to one participant, navigating through the code and editing a specific word on a certain line would be useful.

[1]https://www.nuance.com/dragon.html
[2]https://voicecode.io/details
[3]https://talonvoice.com/

Themes	Description	Illustrative quotes
Experience	Experience of using voice user interfaces to write code	"It's certainly not doing natural language processing in interpreting your commands, but they try to make them feel like a natural command, and I think they do it at the expense of the power and unambiguous nature that you could get." (P1) "A lot of people bounce off of dictating code when they start because it's really annoying and unpleasant to learn all these commands." (P4)
Vision	Vision of a future voice programming system	"I'm going to assume something closer to like the Star Trek computer that has a lot of natural language processing." (P2) "I guess it would be more similar to the experience of pair programming with someone." (P4)
Challenges	Parts of program difficult to input by voice	"Switching between comments and the rest of your code is I think a little hard because those are really two different modes." (P4) "There are some real challenges with variable names that might not be normal spoken words." (P1)
Privacy	Privacy or social concerns of using a voice programming system	"I have privacy concerns about using voice interfaces if I'm not in a private place and can be overheard." (P1) "I would never use a cloud-based speech recognizer to write programs as it's hard to know what information is gathered and who might have access to it." (P6)
Usefulness	Situations in which voice programming would be useful	"People won't be intimidated by typing by hand, so it would give them more energy and focus on actually solving the problem." (P3) "For people who are on the verge of developing RSI and also for people who can't type in the first place." (P7)

Table 2.2: Details of the five themes and representative quotes from the thematic analysis of the interview data.

Theme: Challenges. Six participants thought variable names would be very challenging to dictate as they are sometimes not normal spoken words. Two participants expressed that dictating new variables is hard, and they wish there were a mechanism to save all previously used variable names for faster dictation. Two participants mentioned that switching between dictating programming terms and speaking English was difficult when they needed to write comments in the code. Three participants mentioned that indentation would be a big challenge when writing Python code. All participants pointed out that dictating code from scratch and editing existing code by voice are two distinct things. Five participants mentioned that navigating the code and correcting errors would be challenging.

Theme: Privacy. Two participants were concerned about using a cloud-based recognizer as it's uncertain whether data would be retained and stored. Two other participants mentioned they would be concerned about privacy if they had to speak programs where others could hear them. Others said they would not be concerned about privacy if their voice remained anonymous.

Theme: Usefulness. All participants thought programming by voice would be useful for people with motor impairments like themselves. Two participants stated that programming by voice would also be useful for people with no motor impairments as they can relax while speaking easy parts of a program. Another participant mentioned that programming by voice would be useful for people who program as part of their job as they could focus more on problem-solving. According to all participants, programmers who are at risk of developing RSI or are in the early stages of it need ways to reduce their use of the keyboard

and mouse. For these programmers, voice programming could be very useful in minimizing the strain on their hands and preventing further injury.

2.6 Discussion

The interviews with motor-impaired programmers revealed significant challenges and yielded insights regarding voice programming systems. Most motor-impaired programmers found today's speech recognition technology helpful for general tasks such as writing emails. However, this technology falls short in creating content in markup languages (e.g. HTML, and LaTeX) or writing code in programming languages like Python. In these areas, the available voice solutions are often awkward and require substantial training, forcing users to adapt to the limitations of the technology rather than the technology accommodating their needs.

Learning complex commands emerged as a major hurdle, particularly for novice programmers and those new to programming by voice, such as individuals in introductory courses. Participants emphasized the difficulty in mastering the extensive command sets required by existing systems. This challenge could be mitigated by exploring how programmers speak programs naturally without learning commands.

In theory, programming languages are more predictable than natural languages because they have strict grammar rules. However, this gets complicated by user-defined names such as variable and function names. Variable names may contain abbreviated words and mixed capitalization, which makes them less predictable.

Furthermore, it is not apparent whether programmers will speak a line of code using its exact syntax or more naturally. A key step in developing an intelligent voice programming system would be investigating how programmers speak various programs and user-defined names.

Programming involves more than just writing code; it also includes navigating the code, editing, debugging, and correcting errors. Making these interactions accessible to individuals with diverse motor abilities presents significant challenges because they require precise control and complex commands that can be difficult to perform without traditional input devices. Current voice-based systems primarily focus on code input, but further research is needed to develop effective voice-based tools for debugging and code correction.

While current speech recognition technology provides some benefits for general tasks, it is inadequate for more specialized programming tasks. Addressing the challenges of training data scarcity, command complexity, and user-defined name variability is essential for developing a practical and naturally spoken programming system. Given recent advances in speech and natural language processing, creating such a system appears feasible and would benefit both motor-impaired and able-bodied programmers.

2.7 Conclusion

In conclusion, our interviews revealed a clear desire among motor-impaired programmers for a more natural and purpose-built voice programming solution. De-

spite the availability of several voice programming systems, participants expressed frustration with the current system's accuracy and usability. They highlighted numerous challenges, such as the difficulty dictating code accurately, speaking variable names, switching between coding and commenting, and the lack of comprehensive documentation. Participants also pointed out the steep learning curve of mastering the extensive commands required by existing systems. While there are many challenges, we think a practical and naturally spoken programming system would be valuable to many, given recent advances in speech and natural language processing.

Chapter **3**

Exploring How Programmers Speak Code

As we progress toward developing voice programming systems, it becomes imperative to understand the nuances of how programmers naturally speak code. Training language models for such systems requires large datasets encompassing diverse spoken code examples. This diversity ensures that the models can effectively handle various speech patterns and programming contexts. To collect examples of people speaking programs, we conducted three user studies, each building on the insights of the previous one. In all our user studies, participants spoke to a hypothetical system. Our primary goals were twofold: first, to capture the diverse ways programmers speak code, and second, to create a comprehensive dataset for training language models and fine-tuning acoustic models. Additionally, we aimed to identify effective and scalable methods for collecting spoken programs. This chapter details the methodologies, findings, and implications of our user studies.

3.1 Related Work

Research into voice-based programming systems has evolved over the years, with various studies focusing on different methodologies to facilitate voice programming. This section reviews key developments in voice programming systems, highlighting their methodologies, user studies, and lessons learned from their evaluations.

Arnold et al. [3] designed a command-based voice programming system called VocalGenerator. VocalGenerator takes a Context Free Grammar (CFG) and a voice vocabulary for a programming language as input and generates a programming environment where users can write programs by voice. The system is no longer being developed.

Maloku and Pllana [35] developed HyperCode, a tool that enables coding in Java with voice commands using the speech recognition engine Dragon Naturally Speaking [58]. The tool is embedded within IntelliJ IDEA, a commercial Java Integrated Development Environment (IDE). HyperCode allows users to create their own custom voice commands. In a user study, participants coding with a combination of keyboard, mouse, and voice commands using HyperCode completed tasks faster, with an average time of 46 seconds. In contrast, participants using only a keyboard and mouse took an average of 65 seconds, and those using only voice input took an average of 84 seconds.

Rosenblatt et al. [51] conducted a comprehensive study to develop and evaluate a web application named VocalIDE, a vocal programming editor. A Wizard

of Oz (WOz) study was conducted with ten participants without motor impairments who completed programming tasks by giving vocal instructions to a human controlling a text editor. Based on the results from the WOz study, VocalIDE was designed and developed. VocalIDE enabled users to write and edit code using voice commands. It incorporated a browser-based automatic speech recognition, WebKitSpeechRecognition, and a rule-based syntax parser. VocalIDE was evaluated with eight participants with upper limb mobility impairments to assess its usability. The efficiency of the system was limited by inadequate speech recognition accuracy.

Wagner et al. [66] developed Myna to make block-based visual programming language accessible. Myna is a voice-driven interface designed to enable motor-impaired children to learn to program in Scratch[1]. In an evaluation, Myna took 15.6 seconds less time on average than the mouse and keyboard [65]. However, the study also highlighted a significant challenge: non-native English speakers made more errors while using Myna compared to native English speakers.

Price et al. conducted a Wizard of Oz study to explore a spoken language interface for beginner programmers, simulating a system where students described Java programming tasks aloud. Participants interacted with an expert posing as the system, providing insights into natural language descriptions and preferred vocabulary. The study revealed that untuned speech recognition systems struggled with domain-specific language, and disfluencies in speech posed challenges. Despite these issues, participants responded positively, indicating potential ben-

[1]https://scratch.mit.edu

efits for such a natural language-based interface. However, the simulated nature of the study did not fully reflect real-world interactions, as participants were not interacting with an actual system but with a human intermediary, which could influence their behavior and responses.

Desilets [15] conducted a study to understand the challenges involved in programming by voice, such as dictating punctuation and variable names with abbreviated words and items in mixed cases. The author later developed a tool named VoiceGrip that enabled users to speak code using a pseudo-code syntax that was then translated into native code. The system's capabilities were restricted to a manually created database containing mapping from native symbols (e.g., "<") to pseudo symbols (e.g., `"less than"`) and some predefined rules to understand programming constructs.

Brummelen et al. [63] conducted a user study evaluating a voice-based system called CONVO. CONVO allowed users to program using conversation speech. For instance, when a user says, "Create a program," CONVO responds with, "What do you want to name it?" The authors found that novices appreciated using natural language, while advanced users preferred text input for accuracy and efficiency. This study supports our belief that developing a system allowing users to speak code naturally is worthwhile. However, the reliance on predefined responses and regex-based parsing may limit the system's flexibility in handling diverse programming constructs and languages. The authors also highlighted that current ASR systems are not sufficient for voice-based programming and emphasized the need for a custom ASR model tailored for programming tasks.

Begel and Graham [6, 7] investigated how programmers read aloud Java code written on paper by asking ten expert programmers to read a page of Java code as if they were instructing a second-year undergraduate student. They discovered that, regardless of programming experience, the programmers had similar speech patterns, though their speaking styles varied significantly depending on the programming constructs. This research led to the development of a system called Spoken Java, which uses a rule-based approach to recognize spoken code. This system employs a lexical analyzer to break down spoken commands into tokens and a semantic analyzer to interpret these tokens using contextual information. However, the authors noted potential limitations, suggesting that their method might not capture all variations in spoken code, as people might speak differently when dictating code from scratch versus reading pre-written code aloud. This concern inspired our investigation into collecting spoken code through two methods: one where the speaker can see the code and another where the speaker cannot see the code while speaking.

Few studies have investigated the use of natural language in voice programming. Previous research has primarily focused on developing command-based systems for voice programming, which necessitates that programmers learn specific grammatical structures. Our approach is distinct. We aim to explore how programmers naturally speak code without imposing rigid command structures. Our objective is to understand the diverse ways in which programmers verbalize code, allowing for a more natural interaction with the system.

3.2 Study 1: A Pilot

We began with a pilot study to understand the basic challenges and opportunities in collecting spoken code data. It provided a foundation for refining our methods and ensuring that our subsequent studies would be more effective.

3.2.1 Participants

The pilot study was completed by 16 undergraduate students. Students had an average of two years of programming experience. 43% of participants agreed they frequently wrote programs. 62% of participants were male and 38% were female. All were native English speakers. According to the responses to the initial questionnaire, only 5% participants strongly agreed and 5% agreed that they frequently use voice user interfaces.

3.2.2 Procedure

The pilot study relied on participants downloading and completing the study via a PowerPoint file. Upon signing the consent form, participants were given the PowerPoint file. The file included an initial questionnaire, instructions to complete the experiment, 20 different Java programs, and a final questionnaire. The initial questionnaire included demographic questions, participant's experience of using voice user interfaces, and programming experience. The final questionnaire asked the participants about their overall experience. Participants typed responses into

Java program with line 8 missing

```
1  // Calculates the sum of first n
2  // natural numbers using a for loop
3  public static void main(String[]
       args) {
4     int n, i;
5     int sum = 0;
6     Scanner scan = new Scanner(
          System.in);
7     n = scan.nextInt();
8
9        sum = sum + i;
10    }
11    System.out.println("Sum = " +
          sum);
12 }
```

Java program with line 7 highlighted

```
1  /* The program initializes an
2  * integer variable largeNumCounter
3  * to 0 and increments it.
4  */
5  public class Complex {
6      private double largeNum;
7      private int largeNumCounter=0;
8      public double increment() {
9          largeNumCounter++;
10       }
11   }
```

Table 3.1: Example of two Java programs from the user studies 1 and 2. The left program has a missing line. The right program has a highlighted line.

the slides for the questionnaires. Initial and final questionnaires are included in Section A.2.

We created two PowerPoint files, each having the same 20 Java programs. In the first PowerPoint file, odd-numbered programs had a missing line, while even-numbered programs had a highlighted line. In the second, we reversed this. Half the participants received the first version, and the other half received the second one. The programs involved a variety of statements, including loops, constructors, switches, variable declarations, object creations, if statements, defining and invoking methods, comments, and math statements. One program had a multi-line comment. Each program included comments that helped participants understand the purpose of the program. Table 3.1 shows an example of one Java

program with a missing line and one Java program with a highlighted line. All programs used in the experiment are available online[2].

Participants completed the study remotely using their own computer and microphone. They did not receive any feedback while recording their voice. The participants were paid $15, and some also received extra credit in a course. At the beginning of the experiment, participants were instructed as follows:

> a) "Imagine you are a programmer who has an injury. Typing on the keyboard is difficult for you. How would you speak code to an intelligent computer program that could convert your speech into code?"
>
> b) "There are no rules in how you speak code."

We collected a total of 320 recordings. We trimmed silence and the start and end of recordings from the recorded audio files. Next, we created text transcripts for each audio file. We listened to each audio file and transcribed it verbatim, including spoken symbols, words, and spaces. For example, "`case quotation marks t w o end quotation marks colon`".

3.3 Study 2

Study 1 relied on participants downloading and completing the study via a PowerPoint file. This approach did not allow us to log user actions, limiting our ability to analyze participant behavior during the experiment. In addition, the pilot

[2] https://osf.io/h6nk4

study involved a small, homogeneous group with similar programming experience. This limited our ability to capture a wide range of spoken code utterances. We conducted Study 2 with the goal of understanding: 1) the diverse ways in which different programmers speak code, 2) various ways specific programming constructs can be uttered, and 3) the most effective method for collecting spoken code.

3.3.1 Participants

We recruited 12 novice programmers (7 female, 5 male) from introductory Java courses. We recruited 12 expert programmers (2 female, 10 male) through word-of-mouth. Experts were required to have at least four years of programming experience and be familiar with Java. Experts' programming experience ranged from two to 23 years. All participants were native English speakers. As for their usage of speech interfaces, 8% of novices and 18% of experts strongly agreed or agreed that they frequently used speech interfaces. We asked participants how frequently they wrote programs. 58% of novice participants and all of the expert participants strongly agreed or agreed that they frequently wrote programs.

3.3.2 Procedure

We created two sets of programs, each consisting of 20 identical Java programs. In the first set, the odd-numbered programs had missing lines(s), and even-numbered programs had highlighted line(s). In the second one, we reversed this. Participants were randomly assigned to the first version or the second version. Out

of the 20 Java programs, 16 had single missing or single highlighted lines, while four had multiple missing or highlighted lines of code. The single lines of code included various code constructs, such as function calls, if-else statements, loops, input-output statements, arrays, comments, decrement operations, mathematical calculations, and variable declarations. The multiline code snippets included an if-else block, a for-loop block, a multiline comment, and a function body. Table 3.1 shows an example of a Java program with a missing line and highlighted line as used in our study. On average, the single-line and the multi-line programming statements were 42 and 53 characters long, respectively. All 20 programs were different and ranged from 6 to 16 lines. All programs used in the experiment are available online[3].

Instead of PowerPoint, we used a web application to better log user actions and interactions, providing more precise data. Using the web application, participants first signed a consent form and filled out a demographic questionnaire.

For each program, participants recorded themselves speaking either the missing line(s) or highlighted line(s) of each program. They received no feedback while speaking but could play back their recording afterward. Participants could re-record the audio for a given program as many times as they wanted; we only kept the last recording. Finally, they completed a questionnaire that asked about their experience during the study.

We manually reviewed all collected recordings. Novice participants often submitted empty recordings for the multi-line tasks, which we discarded. In total,

[3] https://osf.io/h6nk4

we collected 224 audio files (192 for single lines, 32 for multi-lines) from the novices and 240 audio files (192 for single lines and 48 for multi-lines) from the experts. We listened to each audio file and typed a verbatim word-by-word transcript of what the person said, including spoken symbols, words, and spaces. For example, `"string very large string two equals quotation mark world end quotation mark semicolon"`. Each recording was transcribed into a one-line transcript for both single and multi-line programming statements. We have made the text transcripts of SpokenJava 1.0 available[4], but unfortunately, we did not have ethics approval to release the audio recordings.

3.3.3 Results

Novices completed the experiment on average in 48 minutes (SD = 8.8), while experts took 45 minutes (SD = 26.2). We analyzed two participant groups, novices, and experts, to analyze two measures: speaking style and speaking rate in both the missing and highlighted conditions. Additionally, we investigated the variations of speaking different programming constructs and the ambiguity in spoken code. Finally, we analyzed participants' subjective feedback.

Speaking Style: Natural versus Literal

Two human judges independently assessed the speaking style of each utterance in the transcripts and categorized each as either natural or literal. Before judging, they discussed the criteria for judgment and the definition of natural and literal

[4] https://osf.io/h6nk4

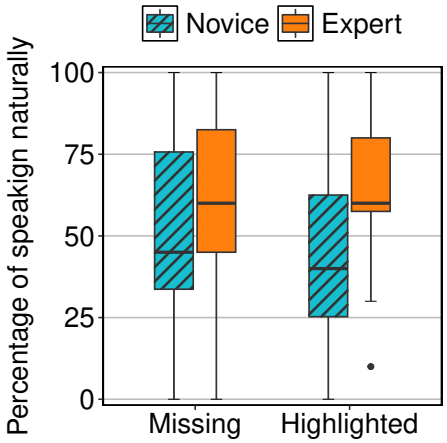


Figure 3.1: Comparison of speaking naturally in the missing and highlighted conditions.

Participant	Condition	
	Missing	Highlighted
Novice	51.1±21.2 [0, 100]	58.2±20.1 [0, 100]
Expert	40.8±20.7 [0, 100]	36.7±17.3 [10, 100]
Mixed ANOVA	Main effect of Condition: $F(1, 22) = 0.2$, $p = 0.65$	

Table 3.2: Numerical results from the user study on the proportion of speaking naturally for different conditions (Missing vs. Highlighted), including statistical test details. Results are presented as mean ± 95% confidence interval [minimum, maximum].

utterances. Utterances were considered natural if participants spoke most parts of the code using natural phrases (e.g., `"start a comment"`) instead of literal adherence to the required characters (e.g., `"forward slash forward slash"`). The two judges did not see each other's ratings beforehand. Inter-rater reliability was very high (Cohen's kappa = 0.98), indicating an almost perfect agreement between the raters. To ensure consistency, the judges then discussed their judgment and resolved any disagreements.

A mixed analysis of variance (ANOVA) design was conducted to investigate

the effects of experience level (novice versus expert) and conditions (missing versus highlighted) on the use of natural language in code dictation. The results revealed no significant interaction between experience level and condition ($F(1, 22) = 3.0$, $p = 0.09$) (Table 3.2). This indicates that the effect of the condition did not differ significantly between expert and novice programmers. Furthermore, there was no significant main effect of experience level on the use of natural language ($F(1, 22) = 1.8$, $p = 0.19$), with expert programmers using natural language slightly more often than novices (62% versus 45%, respectively). However, there was no significant main effect of condition on the use of natural language ($F(1, 22) = 0.2$, $p = 0.65$). These findings suggest that the use of natural language in code dictation is not significantly influenced by experience level and that both novice and expert programmers are similarly affected by the missing or highlighted conditions.

Verbalization by Programming Construct

We found wide variations in how certain programming constructs and some specific parts of code were verbalized. Most variations occurred when speaking method declarations, user-defined names, assignment operations, elements of an array, comments, and punctuation. Fewer variations occurred when speaking if-else statements and for-while loops. Table 3.3 shows some examples of how novice and expert programmers' speech varied.

Method signature and method call Eight expert programmers verbalized different parts of the method, such as return type, method name, and a parameter list naturally (e.g., "`declare function public static return type integer`

Target code	User	Human Transcript
`items[i] = scan.nextInt();`	novice	items square bracket i end square bracket equals scan dot next int semicolon
	expert	items at location i is equal to scan dot next int
`largeNumCounter-;`	novice	large num counter minus minus semicolon
	expert	decrement large num counter
`while(num >= 1)`	novice	while parenthesis num is greater than or equal to one end parenthesis quotation
	expert	start while loop start condition num is greater than or equal to one end condition end while loop
`total = calculate_sum(age,5);`	novice	total equals calculate underscore sum open parenthesis age comma five close parenthesis semicolon
	expert	variable total is equal to method calculate sum where the first argument is variable age and the second argument is the number five
`// this program searches an array for the minimum value`	novice	comment this program searches an array for the minimum value end comment
	expert	start comment this program searches an array for the minimum value end comment

Table 3.3: Some examples of the variations in the speech of novice and expert programmers.

`name cube parameter int num`"). In contrast, all but one novice programmer spoke methods in a literal way (e.g., "`public static integer cube open paren int num close paren`").

User-defined names While dictating user-defined names such as variable and method names, two expert programmers mentioned naming conventions (e.g., "`camel case very large string`") while one expert spelled them out. Five other experts and three novices verbalized capitalization (e.g., "`large capital n num capital`

`c counter`"). We also observed that two expert programmers, but no novices, occasionally said the term "`variable`" while dictating a variable name.

Comments As comments are written in natural language that might involve references to variable names or method names such as "`// The method getMin searches an array for the minimum value`". We wanted to see how participants spoke such elements in the comment and how they switched between the syntax required to denote a comment and the comment itself. All experts but only 4 novices started a comment by speaking "`open comment`", "`header comment`" or "`comment`". The other novice participants spoke comments by verbalizing "`slash slash`" or "`forward slash forward slash`". Nine experts explicitly mentioned if their intent was a single-line or multi-line comment.

Abbreviated words Abbreviated words were either spoken as full words or spelled out. 80% of experts and novices verbalized the function "`sqrt`" as "`square root`" while others spelled it out. Two expert programmers verbalized the full form for an abbreviated variable name, for example, saying "`number`" instead of "`num`". Additionally, 90% of the experts and 40% of the novices verbalized the function "`println`" naturally as "`print line`" while others spoke it as "`print l n`".

Assignment operation 70% of experts used phrases such as "`is assigned`" or "`becomes`" or "`set`" instead of verbalizing the "`equal`" symbol. for instance, one expert uttered the variable assignment "`i = 1`" as '`i is assigned one`". None of the novices used such natural phrases for assignment operation.

Multi-line code We had programmers speak four multi-line programs. We aimed to explore how participants might verbalize a block of code, including, for

example, specifying the transition to a new line. Most experts uttered phrases like "new line", "enter", "begin body", or "start for loop body" to transition to a new line. None of the novices explicitly uttered any term for a new line. Instead, they dictated the entire code block as if it were on a single line (e.g., "static int cube int num end parenthesis curly bracket return num times num times num curly bracket").

Symbols and punctuation

We found significant variations in spoken punctuation. Experts who spoke naturally used some natural phrases for punctuation; for instance, two experts uttered "end line" instead of "semicolon". In addition, eight experts and three novices dictated array elements as "items at location i", "items at index i" or "items sub i" while the other participants dictated them in a literal way (e.g., "items open square bracket i close square bracket").

Participants used a variety of terms to refer to the quote symbol, including natural terms like "character" or "string" as well as more specific terms like "quote", "single quote", "opening quote", "end quote", "quotation marks". Additionally, participants spoke parentheses in varied ways, such as "paren", "left parenthesis", "open parenthesis", and "close parenthesis". Variation also occurred in speaking brackets or braces, e.g., "left curly brace", "open curly brace", "close curly brace", "curly bracket" or "bracket".

Participants who spoke in a literal way had a tendency to omit punctuation in both highlighted and missing conditions. We considered all single-line literal

utterances and calculated the proportion of spoken punctuation in the transcript to actual punctuation in the target code. Notably, only participants who spoke literally at least 50% of the time in both conditions were included in the analysis, consisting of eight novices and three experts.

Overall, participants spoke parentheses 83% of the time in the highlighted condition versus 72% of the time in the missing condition. Interestingly, participants verbalized quotation marks 100% of the time in the highlighted condition but only 42% of the time in the missing condition. In the case of semicolons, participants spoke fewer in the missing condition (60% of the time) compared to the highlighted condition (69% of the time). This suggests two potential explanations for the differences in punctuation use. First, it is possible that participants struggled to balance out the parentheses or quotes when they could not see the line of code, leading to a decrease in their use of punctuation. Second, it is possible that participants anticipated that an intelligent voice programming system would auto-complete the missing punctuation, leading them to rely less on their own use of these punctuation marks.

Correctness and Semantic Ambiguity

We suspected participants might sometimes speak incorrect code (i.e. code that does not achieve the program's stated objective). This could occur especially often when participants could not see the line. Missing lines could be problematic because participants had to infer the missing content based on the surrounding code, which could result in incorrect or incomplete utterances. Even highlighted

lines could result in incorrect utterances if participants misunderstood the context or omitted something important. For example, consider a highlighted math statement: `"double result = (a + b) * c;"` A participant might incorrectly interpret and speak: `"double result equals a plus b times c"`, changing the intended order of operations and potentially leading to incorrect results.

Two human judges independently categorized each spoken single line of code as either correct or incorrect. we looked at the whole program since we needed to judge if it would result in a correct implementation. Utterances were considered incorrect when the spoken code was incomplete, incorrect, or ambiguous. Inter-rater reliability was high (Cohen's kappa = 0.88), indicating close agreement between the raters. To ensure consistency, we reviewed our ratings and resolved any disagreements.

We calculated the proportion of participants' incorrect spoken code. Overall, novices spoke incorrect lines 16% of the time, while experts spoke incorrect lines 7% of the time. As might be expected, participants spoke more incorrect code in the missing condition than in the highlighted condition. Novices spoke incorrect lines 28% of the time in the missing condition but only 4% in the highlighted condition. Similarly, experts spoke incorrect lines 12% of the time in the missing condition but only 2% in the highlighted condition.

We observed that the incorrect spoken programs in the highlighted condition were always the result of unclear or ambiguous speaking patterns. We felt a voice programming system might have difficulty accurately transcribing such speech. For example, a few participants spoke the line of code "`result=Math.sqrt(x+y)/z`"

as "result equals math dot square root x plus y divided by z". Without mentioning the order of arithmetic operations, the system might interpret this as "x+y/z". Some participants mentioned the order explicitly, for example, saying "result equals math dot square root open paren x plus y close paren divided by z".

Some participants failed to specify whether a line of spoken code included a digit or a character. This particularly occurred in the conditional statement "c <= '9'" in which participants simply spoke it as "if c less than or equal to nine". We also observed a few participants mistakenly spoke "backslash" while dictating a comment instead of "forward slash". Additionally, we observed that some participants did not specify whether a line of spoken code was a comment but instead spoke just the comment's text. Such ambiguity or lack of context may lead to an inaccurate machine translation of the spoken code to its target code.

Speaking Rate

We trimmed silence from the start and end of the recordings and calculated the speaking rate of an utterance in words per minute (wpm). As we did not have enough multi-line code from novices, we analyzed only the single lines. In the transcripts, there were about 18.3 words per missing line and 19.2 words per highlighted line on average.

A mixed ANOVA analysis revealed no significant interaction between experience level and condition ($F(1, 22) = 0.2$, $p = 0.67$) on speaking rate. There was no significant main effect of experience level on the speaking rate ($F(1, 22) =$

Participant	Condition	
	Missing	Highlighted
Novice	84.0±10.1 [55.9, 114.0]	92.7±10.6 [75.2, 125.0]
Expert	95.2±16.7 [44.5, 138.0]	105.8±16.3 [59.8, 148.1]
Mixed ANOVA	Main effect of Condition: $F(1, 22) = 18.9$, $p = 0.0003$	
Pairwise t-test (Novice)	Novice: Highlighted > Missing, $p = 0.025$	
Pairwise t-test (Expert)	Expert: Highlighted > Missing, $p < 0.001$	

Table 3.4: Numerical results from the user study on speaking rate under different conditions (Missing vs. Highlighted), including the statistical test details. Result format: mean ± 95% CI [min, max]. The speaking rate is measured in words per minute (wpm).

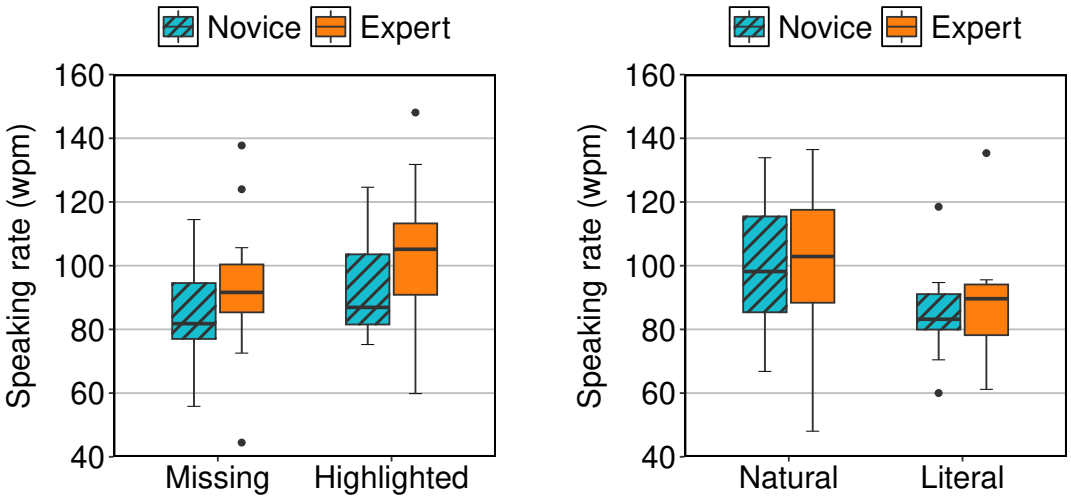
1.20, $p = 0.29$). However, the condition had a significant effect on the speaking rate ($F(1, 22) = 18.9$, $p = 0.0003$). Post-hoc pairwise comparisons with Bonferroni corrections revealed that experts spoke significantly faster in the highlighted condition than in the missing condition ($p < 0.001$), with 105.8 wpm in highlighted versus 95.2 wpm in missing. Similarly, novices spoke significantly faster in the highlighted condition ($p = 0.025$), with 92.7 wpm compared to 84.0 wpm in the missing condition (Table 3.4). It might be the case that the increased cognitive demands of mentally visualizing the target line based on the surrounding code may have required additional time, resulting in a slower speaking rate when participants could not see the line.

We also calculated the speaking rate of novices and experts when speaking naturally versus when they spoke in a literal manner. We excluded participants who always spoke naturally (one novice and two experts) or always spoke in a literal manner (one novice). This resulted in a sample of ten novices and ten experts. We found no significant interaction between experience levels and speaking styles on

Participant	Style	
	Natural	Literal
Novice	98.9±14.3 [66.8, 133.9]	85.1±10.3 [60.0, 118.5]
Expert	99.5±17.6 [48.0, 136.4]	89.1±13.0 [61.2, 135.3]
Mixed ANOVA	Main effect of Style: $F(1, 18) = 9.231$, $p = 0.007$	
Pairwise t-test (Novice)	Novice: Natural > Literal, $p = 0.02$	
Pairwise t-test (Expert)	Expert: not significant, $p = 0.14$	

Table 3.5: Numerical results from the user study on speaking rate under different speaking styles (Natural vs. Literal), including the statistical test details. Result format: mean ± 95% CI [min, max]. The speaking rate is measured in words per minute (wpm).

the speaking rate ($F(1, 18) = 0.2$, $p = 0.68$) (Table 3.5. There was no significant main effect of experience level ($F(1, 18) = 0.1$, $p = 0.79$), but there was a significant main effect of speaking style on the speaking rate ($F(1, 18) = 9.2$, $p = 0.007$). Post-hoc pairwise comparisons with Bonferroni corrections indicated that novices had a significantly faster speaking rate when speaking naturally than speaking literally ($p = 0.02$). At the same time, there was no significant difference in speaking rate between the two styles for experts ($p = 0.14$). This suggests that the effect of speaking style on speaking rate did not differ significantly between novice and expert speakers, but how novice participants spoke affected their speaking rate. It might be the case that following strict rules of grammar and syntax imposed additional cognitive demands on novices which slowed down their speaking rate compared to a more natural speaking style. Additionally, syntax-style utterances could likely be harder to say fluently.



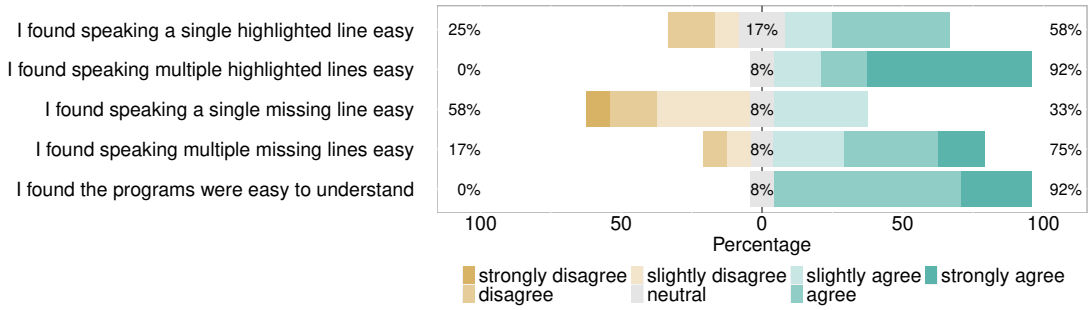
(a) Missing versus highlighted **(b)** Natural versus Literal

Figure 3.2: Comparison of speaking rate.

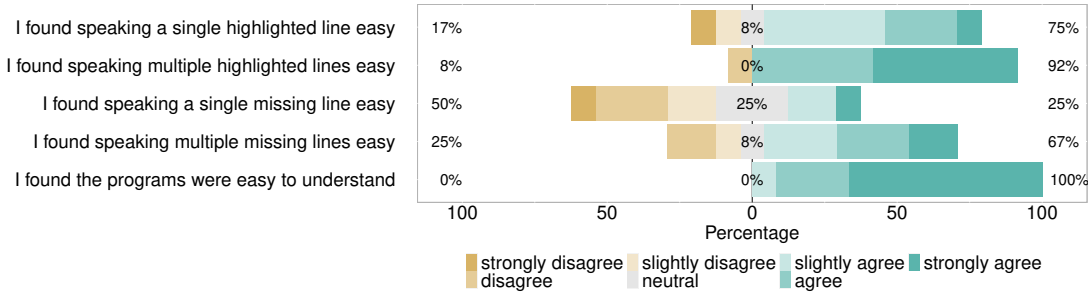
Subjective Feedback

Participants rated five statements about their overall experience with the study on a 7-point Likert scale (1=strongly disagree, 7=strongly agree). Experts' ratings significantly differed ($\chi^2(3) = 111.3$, $p = 0.00001$). Posthoc pairwise comparisons with Bonferroni adjustment revealed that experts found it significantly easier to speak a single highlighted line (mean = 6.2, SD = 1.4) compared to a single missing line (mean = 4.8, SD = 1.6) ($p = 0.0002$) with a large effect size (Cohen's d = 0.89). There were no significant differences in their ratings for the ease of speaking a single missing line versus multiple missing lines (mean = 3.5, SD = 1.7) or between a single highlighted line versus multiple highlighted lines (mean = 4.8, SD = 1.8).

A significant difference was also found in novices' ratings ($\chi^2(3) = 121.7$, $p <$



(a) Feedback from novices



(b) Feedback from experts

Figure 3.3: Novice participant feedback (top) and expert participant feedback (bottom). The percentages on the left are the portion of participants who strongly disagreed, disagreed, or slightly disagreed with the statements. The percentages in the middle correspond to the portion who were neutral. The percentages on the right correspond to those who strongly agreed, agreed, or slightly agreed

0.000002). Post-hoc pairwise comparisons revealed significant differences in their ratings for the ease of speaking a single highlighted line (mean = 6.2, SD = 1.4) versus a single missing line (mean = 4.8, SD = 1.6) ($p = 0.036$) with a large effect size (Cohen's d = 1.25), as well as for the ease of speaking multiple highlighted lines (mean = 5.2, SD =1.5) versus multiple missing lines (3.4, SD = 1.4) (p = 0.036) with a large effect size (Cohen's d = 1.20). However, there were no significant differences in their ratings for the ease of speaking a single missing line

versus multiple missing lines or between a single highlighted line versus multiple highlighted lines.

In general, both novices and experts found speaking a missing line challenging, but novices faced more difficulty in speaking multiple missing lines. One expert said, "It is more difficult to make nontrivial code from scratch as opposed to reading a line that already exists". Speaking missing lines of code seems practical and relevant as in the context of an actual voice programming system, people may need to describe code without being able to see it. However, verbalizing missing code does require more cognitive effort. Although we provided participants with some context in the form of comments (e.g., "this program reads ten integers from standard input into an array named items") in both highlighted and missing conditions, it is worth noting that such comments might lead participants to rely heavily on the provided information. One expert participant acknowledged that he was biased, "When reading comments, I had a temptation to want to follow the comment that was provided". This suggests that an effective data collection methodology for spoken code needs to balance the benefits and drawbacks of speaking missing lines to ensure reliable data while also taking steps to avoid biasing the programmer to one particular solution.

We asked participants about the ease of the programs. All experts and all but one novice found the programs easy to understand. When asked about the parts of programs they were most uncertain about how to dictate, novices and experts had differing opinions. All experts and three novices indicated that dictating function declarations was the hardest. One expert said, "I was most uncertain

about method keywords, for example, how to differentiate different parts of the method declaration.". None of the novices but five expert programmers thought dictating variables was really challenging. One expert said, "It's complicated to figure out how to speak variable names when considering issues like capitalization and whether to spell out an identifier.".

Some participants were uncertain about whether to dictate punctuation, and they believed that an intelligent voice programming IDE should handle punctuation automatically, especially when it comes to balancing braces and parentheses. According to a novice programmer, "I was uncertain mostly what punctuation I needed to state directly and what could be auto-completed". This suggests further investigation is required to overcome the challenges in dictating difficult parts of code such as method declarations, variable names, and punctuation.

A few expert programmers shared additional comments on how an intelligent voice programming tool should work in general. One expert said "I started off very literal but pretty soon realized that would be an unmanageable way to code and started assuming a smarter model. For instance, typically, Java style is to camel case variable names, so I assumed that should be the default interpretation". Another expert programmer noted, "There is a lot of nuance to simple code such as Math.sqrt(x+y). Although it's very short and simple to spell out, I found it really challenging to try to express it in a command-type way". These insights support our approach of collecting data by asking participants to speak code without imposing any rules, enabling the system to account for the diverse ways in which people might speak code. For instance, an intelligent system should

be able to recognize the function `"sqrt"` regardless of whether it is spelled out letter-by-letter or spoken naturally as `"square root"`.

3.4 Study 3

Based on the insights from Study 2, we conducted Study 3 with the aim of constructing a dataset containing a wide range of programming statements robust enough to train language models and creating a publicly available dataset that other researchers could use. In Study 2, we observed variability in how different programmers spoke code, with different participants verbalizing the same line of code in multiple ways. This variability highlighted the need for a dataset that captures a diverse set of programming constructs and spoken utterances. To address these issues, we made several changes in Study 3. Each participant spoke a variety of statement types and completely different lines to capture a wide range of programs spoken by different speakers. Additionally, we only considered highlighted lines based on feedback from Study 2, where participants found it challenging to infer and speak missing lines accurately. We also observed that participants spoke highlighted lines faster and with similar variability as they did for missing lines.

3.4.1 Participants

We recruited 28 programmers (3 female, 25 male) through programming courses and word of mouth. Participants' ages ranged from 18 to 56 years, and their programming experience varied from 2 to 40 years. All participants were na-

tive English speakers. Among them, 29% strongly agreed or agreed that they frequently use speech interfaces to control a computer. A significant majority, 91.4%, strongly agreed or agreed that they frequently write programs, but none used voice-to-input programs. We also inquired about disabilities; one participant reported having carpal tunnel syndrome. The exact questionnaire used is available in Section A.3.

3.4.2 Data Extraction and Preparation

To generate the lines of code for our study, we extracted Java code snippets from the CodexGlue [34] dataset. CodexGlue is derived from CodeSearchNet [26] and includes a filtered set of examples that meet specific criteria: they must be parsable into an abstract syntax tree, contain between 3 and 256 tokens, and be in English. The CodexGlue dataset contains both code snippets (a method) and their corresponding method header comments. For our study, we extracted the Java code snippets, each representing a single method. We then used the JavaLang parser to extract and construct different programming constructs from these snippets, such as for loops, while loops, if-else statements, method signatures, method calls, mathematical statements, variable declarations and initializations, and comments. Additionally, we extracted any code before and after the line to provide context for our machine translation experiment described in Chapter 5. An example from our dataset is shown in Table 3.6

Human Transcript
for int j equals offset while j less than length j plus plus

Target line of code

```
for (int j = offset; j < length; j++)
```

Preceding code

```
public void processData(int offset, int length, byte[] data) {
    if (data == null || offset < 0 || length > data.length) {
        System.out.println("Invalid data or parameters.");
        return;
    }
```

Following code

```
    processByte(data[j]);
}
System.out.println(offset + " to length " + length);
```

Table 3.6: An example from the SpokenJava 2.0 dataset.

3.4.3 Procedure

From Study 2, we observed that using highlighted lines could be a faster and more effective way to capture sufficient variations, similar to missing lines. Therefore, in Study 3, we adopted the approach of only providing highlighted lines. Each participant was given 50 lines of code and asked to read each line aloud. In Study 2, participants were given the same programs and spoke the same lines of code, which resulted in overlapping data. This overlap was not ideal for training models, as it reduced the variety and size of the dataset. This time, we ensured that each participant received 50 different lines to maximize the variety of programming constructs collected. Participants were given the same instruction prompt as in

studies 1 and 2, emphasizing that there were no specific rules for speaking the code.

We also noted significant variability in loops, if-else statements, variable declarations, and methods from Study 2. Thus, we allocated 20% of the lines to each of these constructs per participant. For other types of statements, such as comments and mathematical statements, we allocated 10%. The variable declaration and initialization statements were distributed to cover different types, including float, double, int, boolean, and string.

3.4.4 Data Analysis

Unlike previous studies, which focused on detailed speaking style and preferences analysis, the primary motivation of this study was to generate a comprehensive dataset while also gathering additional subjective feedback to understand user preferences and improve future voice programming systems.

We first listened to each audio file and transcribed them verbatim, capturing spoken symbols, words, and spaces, just as we did in the previous study. Incomplete or empty audio recordings were filtered out to maintain the dataset's quality. In total, we collected 1243 spoken single lines of code from 28 speakers. The final dataset includes audio files, their corresponding transcripts, before-and-after code snippets, and the target code. The dataset is available online[5].

Then we conducted a thematic analysis of participants' responses to understand their experiences during the study, preferences, and vision for an ideal voice

[5] https://osf.io/6axd2/?view_only=518e06e29e1e4e0b90a9447e6e56c84f

programming system. This analysis aimed to understand their experiences with the study, preferences, and vision of an ideal voice programming system. Two human judges independently reviewed the responses and identified themes. Discrepancies were resolved through discussion with a third reviewer, and the final themes were agreed upon.

3.4.5 Subjective Feedback

The thematic analysis revealed the following key themes:

Theme: Preference for flexibility versus rules

Participants were asked, "Did you find it preferable to speak with no strict rules, or would you rather learn specific rules for spoken commands?" There was a mix of preferences for specific rules for precision or a more natural, rule-free approach.

In our study, we did not impose strict rules on how participants should speak the code, like Study 1 and Study 2. We wanted to observe their natural speaking tendencies and to gather insights into their preferences for a more intuitive voice programming system. Based on the responses, 15 participants preferred to speak without strict rules. One participant stated, "I found it preferable to speak with no strict rules as it would be more accessible to talk". In contrast, eight participants expressed a preference for learning rules. One participant noted, "I would rather learn specific rules as long as they were decently intuitive so that I could get a more precise outcome".

Additionally, five participants appreciated the freedom of speaking without strict rules but mentioned that having rules for specific constructs could be ben-

eficial. For example, one participant noted, “Speaking without strict rules feels more natural, but if common commands (loops and method declarations) had set spoken syntax, there would be less confusion on what is actually going to be declared”. Another participant added, “I think a spoken command for things like ’system.out.println’ could be perfect or for creating a Java Main method would be very useful”. Others did not clearly indicate a preference for flexibility or rules.

Theme: Challenges in verbalizing programming constructs

Participants were asked, “What types of code (e.g., variables, loop, method) were you most uncertain about what to speak and why?” Participants reported experiencing significant challenges when speaking method headers, variable names, loops, and mathematical statements.

Most reported difficulty speaking complex code constructs such as method headers because they have multiple components, including access modifiers, return types, method names, and parameters. One participant expressed, “Writing out a method header seemed to be the most uncertain as there are many different parts to it, and if it was more complex, it would be harder.”

Variable names also posed a significant challenge due to the need for careful verbalization of cases and different naming conventions, such as camel case and snake case. A participant noted, “Variable names were tough because you have to be careful with the capitalization and underscores.” This statement reflects the complexity of maintaining accurate case sensitivity and understanding Another participant highlighted this issue, saying, “Longer variable names with multiple words can be really tricky to say without making a mistake”.

Math statements presented another layer of complexity due to the need to clearly convey sequences and relationships among variables, operators, and numbers. One participant said, "The math statements were difficult since they commonly had a group of letters and numbers that relied on stacking multiple operations."

Loops were another source of uncertainty, particularly because of the need to articulate initialization, condition, and increment parts of the construct. One participant explained, "I found loops to be difficult because of the semicolon-separated code and the incremented variables."

Punctuation and syntax, such as parentheses, introduced challenges. A participant commented, "The code with lots of parentheses got a little confusing on when to specifically mention them or not." This quote highlights the difficulty of remembering and accurately expressing the correct use of punctuation in spoken code.

Theme: Vision for a future voice programming system

Participants were asked, "Imagine you are writing a fairly complex line of code. How would you envision a speech interface working to support entering such a line?" The responses indicated a strong desire for a speech interface that could interpret code contextually while also providing real-time feedback.

Participants envisioned a more intelligent and context-aware system. One participant shared, "I imagine that it would work by interpreting what exactly is being called (function, if statement, etc.) and then looking for code that could fit into each parameter." Another participant emphasized the importance of context-

aware commands, stating, “Having a system that understands the context of what I’m trying to do, like setting up an if-statement or a loop, would make things much smoother.” This highlights the necessity for the system to grasp the broader context of the user’s coding task to provide relevant assistance.

Participants showed a desire for the system to break down and understand complex conditional logic step-by-step. One participant suggested, “It may be most efficient to start by identifying the main if statement with its four ’and’ conditions first. Then, move to the first ’and’ condition. If that condition has an ’or’ condition inside it, the system should recognize this and allow you to specify each part of the ’or’ condition one by one.” This implies a desire for the system to follow a clear and structured way of inputting code, where users can speak each part of the code step-by-step.

Additionally, participants suggested that real-time feedback could significantly enhance the usability and efficiency of a voice programming system. One participant mentioned, “It would have to be able to write as the person spoke so they could catch any possible mistakes before it writes a whole line wrong.” This means that the system should immediately transcribe spoken code, allowing users to see and correct errors instantly rather than waiting until a full line of code is completed.

Overall, the thematic analysis revealed a diverse range of preferences for flexibility versus specific rules in voice programming, significant challenges in verbalizing complex code constructs like method headers, variable names, loops, and mathematical statements, and a strong desire for a context-aware, interactive sys-

tem with real-time feedback. These findings highlight the need for developing an intelligent, adaptive voice programming tool that enhances usability and efficiency while catering to various user preferences and addressing the specific challenges faced during code verbalization.

3.5 Discussion and Limitations

The primary goal of our work was to understand how programmers naturally verbalize code to develop voice programming systems that reduce the need for memorizing commands, making programming more accessible and intuitive. Our user studies aimed to capture the diverse ways programmers speak code and the various methods for collecting spoken code data. Rather than imposing a prescriptive grammar for writing such statements, we observed how programmers naturally speak code. This approach, if successful, could ease or eliminate the need for programmers to remember complex commands.

In Study 2, participants spoke code both naturally and literally. This aligns with the trend of using natural language prompts for code generation, making programming more intuitive. However, natural language can introduce ambiguity. For instance, a prompt like "find the most similar items in a collection" is unclear without further context. On the other hand, literal speaking, while potentially more tedious, can increase accuracy since it adheres more closely to the exact syntax required. Another significant finding from Study 2 was that both novice and expert programmers did not show different speaking styles in the missing

versus highlighted conditions, but they spoke faster in the highlighted condition. This suggests that the reduced cognitive demand in speaking presented code, as opposed to generating code from scratch, contributed to this difference.

Existing code corpora, such as CodeXGLUE and CodeSearchNet, are sourced from written code. While these corpora are useful for tasks like code summarization, code suggestion, and code generation, they are not well-suited for training systems to transcribe spoken code line-by-line. Motivated by the need for a dataset that can support the development of voice programming systems, we conducted User Study 3 to address this gap. In this study, we collected data that includes spoken single lines of Java code, their corresponding transcripts, before-and-after code snippets, and the target line code. This dataset can be used for tasks like code generation and spoken line-by-line program recognition. The surrounding context can provide better support for code generation, enhancing the accuracy and usability of such systems. To our knowledge, this is the first work to build a spoken program corpus of this nature, making a significant contribution to the field of voice programming research. However, scaling up from our initial dataset presents several challenges. Recruiting participants with programming experience is necessary but difficult due to the specialized nature of the population.

One limitation of our current approach was having people record themselves speaking code to a hypothetical system. It could be that a person's speaking style changes when interacting with a real system. Lacking a real system, one could instead collect audio via a Wizard of Oz approach. However, a more significant challenge is collecting a larger and more diverse data set. This data should include

programmers with motor impairments, a wider range of programming constructs, and languages beyond Java. While many languages share similar programming constructs, the details of a specific language may require additional support for certain purposes (e.g., how to specify indentation in Python).

Another limitation is that we asked programmers to speak single lines of code in our user studies. People may speak differently when creating an entire program from scratch, working on more complex programs, or using advanced language constructs. While generating blocks of code from a single utterance (e.g., "create a for-loop that prints all the prime numbers in array nums") might be desirable, it may not align with all programmers' needs and preferences. Novice programmers or experienced programmers may prefer speaking single lines of code for editing or modifying existing code rather than generating large blocks of code from a single utterance. Supporting robust entry of individual lines is a necessary first step before considering the input of larger blocks. Even single lines of code can be long, requiring an interface incrementally displaying a partially completed line as the user speaks.

3.6 Conclusion

This chapter has laid the groundwork for developing voice programming systems by investigating how programmers naturally verbalize code. Our user studies captured the diverse ways in which programmers speak code, moving away from prescriptive grammar and instead observing speech patterns without imposing

any rules. The findings from our studies inform the development of an intelligent, adaptive voice programming tool that enhances usability while catering to various user preferences for speaking naturally or literally, addressing challenges in code verbalization, and incorporating natural language processing for handling ambiguities and contextual understanding effectively. We developed a spoken program corpus that can be a valuable resource for developing a voice programming system and can contribute to the research community. Expanding our dataset to include a more diverse range of participants, additional programming constructs, more complex constructs, and multiple programming languages will further strengthen the applicability of a voice programming system.

The subsequent chapters will build upon the foundation established here. Chapter 4 will focus on the first step of our process, which involves speech recognition experiments and detailing how our dataset is used to improve the accuracy of recognizing spoken code. Chapter 5 will explore the second step, which involves machine translation techniques for converting recognized speech into accurate code.

Chapter 4

Recognizing Spoken Programs

4.1 Introduction

The speech patterns among different programmers and programming constructs vary significantly compared to natural language. For example, programmers might verbalize the assignment statement `int x = 10;` in a natural way such as "`declare an integer x and set it to ten`", or they may choose a more literal approach such as "`int x equals ten semicolon`". This distinctive dialect and hybrid language style and limited data in this domain make accurate recognition challenging.

This chapter focuses on the first step in our two-step pipeline: recognizing the literal words spoken by programmers. We initially investigate the effectiveness of a commercial speech recognizer and then train a research-based recognizer using our collected data, aiming to tailor the recognition process more closely to the specific nuances of spoken programs.

4.2 Related Work

To the best of our knowledge, there is no existing work in the literature that specifically adapts commercial or research speech recognizers to recognize spoken programs accurately. While systems like Dragon Naturally Speaking [58] have been used for programming tasks, they have not been adapted to fully address the unique challenges of recognizing spoken programs. The closest related work involves ASR for low-resource domains, focusing on developing effective recognition systems despite limited training data. In this section, we discuss works related to Automatic Speech Recognition (ASR) for low-resource domains, speech recognition in domains with strict syntax similar to programming languages, and advancements in language models for code generation.

4.2.1 Speech Recognition in Low Resource Domains

The traditional approach for ASR relies on a supervised method where ASR models are trained with a large number of audio examples paired with their corresponding transcriptions. One of the major challenges of building an ASR system for a specific domain is the need for ample labeled training data. In addition, transcribing such audio data is time-consuming and labor-intensive. This led to the development of semi-supervised and unsupervised approaches to leverage abundant unpaired audio and text data [4, 5, 71]. Self-training for end-to-end speech recognition [28] is another approach that uses noisy labels generated from a model trained on a smaller labeled dataset. In self-training, a model initially trained on

a smaller labeled dataset generates transcriptions (noisy labels) for a larger set of unlabeled audio data. These generated transcriptions are then used as training data to improve the model further. This method helps overcome the scarcity of labeled data by using the vast amounts of available unlabeled data.

Other approaches include predictive coding [14], which focuses on learning transferable speech representations that can be applied to various downstream tasks. For example, a model trained with predictive coding can be fine-tuned for tasks like speech recognition, speaker identification, or emotion detection. Fine-tuning involves taking a pre-trained model and further training it on a specific task to improve its performance in that domain.

Weak distillation [31] emphasizes improving performance in recognizing rare words and proper nouns. For instance, weak distillation techniques can help an ASR system better recognize uncommon names or technical jargon not frequently encountered in the training data.

4.2.2 Speech Recognition in Syntax-Intensive Domains

Although the recognition of spoken programming languages has not been explored, notable progress has been made in similar fields requiring precise syntax and specialized symbols, such as mathematics and SQL queries. For instance, Song et al. [57] developed an end-to-end neural architecture named SpeechSQLNet to translate human speech into SQL queries. In the mathematics domain, tools such as Math Speak & Write [23], MathSpeak [56], and TalkMaths [67] convert speech into mathematical equations. These tools enable users to dictate mathematical

expressions, making the process more accessible and efficient for those who cannot use traditional input methods.

4.2.3 Advances in Code Generation Models

In recent years, large language models have made significant strides in generating code from text. Code generation models such as CodeBert [19], CodeGPT [34], and PLBART [1] primarily focus on single-turn code generation, where users express the intent of a complete block of code (e.g., an entire function) in one utterance. These models are trained on extensive corpora of code and corresponding comments sourced from open repositories like GitHub. Specifically, CodeBert was trained on the CodeSearchNet [26] dataset, which pairs method header comments with a single method. In contrast, CodeGPT and PLBART were trained on the CodeXGLUE [34] dataset, a filtered version of CodeSearchNet.

Nijkamp et al. [38] developed CODEGEN, a model capable of program synthesis, which entails generating executable code from comments. They demonstrated that multi-turn program synthesis could effectively enhance program synthesis quality by using comments intended for subprograms. The CODEGEN models were trained on three datasets: THEPILE, BIGQUERY, and BIGPYTHON. THEPILE [21] is an 825.18 GiB English text corpus created for language modeling. BIGQUERY, a subset of Google's publicly available BigQuery dataset[1], contains code in multiple programming languages. BIGPYTHON [38] focuses on Python programming language data and is compiled from public, permissively

[1]https://cloud.google.com/bigquery/public-data/

licensed Python code on GitHub as of October 2021.

While existing models leverage comments that programmers write to express the functionality of a section of code, these comments might lack detailed information or fail to align perfectly with the written code. For instance, a comment might describe the purpose of a function without detailing the specific implementation steps, causing the ASR system to produce incorrect or incomplete code. Another significant challenge is the limited availability of spoken programming language data compared to widely available general speech data, such as LibriSpeech (960 hours) [46] and Common Voice (over 9,000 hours) [2]. Programmers' speech includes domain-specific terms, user-defined names, and abbreviated words not commonly found in everyday language corpora. Collecting and transcribing this type of data is time-consuming and costly, as it requires programming expertise. Despite these challenges, code generation models that understand comments and are familiar with code-related terms can be integrated into the ASR pipeline for more accurate transcription of spoken programs. Additionally, leveraging existing comment datasets and utilizing the structured information in written comments can improve the ASR system's understanding of programming-specific language and syntax, thereby bridging the gap and enhancing transcription accuracy.

4.3 Experiments with A Commercial Recognizer

We conducted offline recognition experiments on our collected data with Google Cloud Speech-to-Text[2]. We used our SpokenJava 1.0 dataset as test data: 224 recordings (after discarding the incomplete recordings) from the novice study and 240 recordings from the expert study as described in Chapter 3. The aim was to establish a baseline for comparison with the research recognizer later.

4.3.1 Experiment Setup and Methodology

First, we recognized our audio recordings using a web API by submitting audio using 16-bit linear encoding at a sampling rate of 16 kHz. Google Cloud Speech-to-Text provides a language model adaptation feature, which improves recognition accuracy by incorporating custom transcripts that help the model understand the context better. We used Google's language model adaptation feature, incorporating human-generated transcripts.

We performed a leave-one-participant-out cross-validation approach, a modified form of k-fold cross-validation where each fold excluded one participant's data. In each fold of this approach, we adapted the model using data from all participants except one and then tested it on the excluded participant. This method ensured that the model was tested on speech from unseen users. This process was repeated for each of the 24 participants.

To avoid overlap between adaptation and test data, we split the transcripts

[2] https://cloud.google.com/speech-to-text

into two sets: the first half containing transcripts from the first ten programs and the second half from the last ten programs. Each set included distinct programming constructs such as loops, if-else statements, expression statements, arrays, comments, and functions.

Google's model adaptation requires a set of unique phrases to improve recognition accuracy. We generated different n-grams from the adaptation transcripts, ranging from bigrams (e.g. "left paren") to 5-grams (e.g. "for i equal to zero"). These n-grams were chosen to meet the limits of Google's speech-to-text adaptation API (5000 maximum phrases per request, 100,000 characters per request, and 100 characters per phrase). We found that bigrams yielded the best recognition accuracy, likely because frequently repeated phrases or keywords like "system dot", "int i", "plus plus", and "open paren" are more common in spoken code transcripts than complete sentences. Higher-order n-grams detect more complex and infrequent patterns and increase the model's complexity and computational cost, which can be impractical for larger datasets. A detailed breakdown of WER for all n-grams is shown in Table B.1.

To investigate the impact of the data collection strategy, we adapted the model using transcripts from either the missing lines or the highlighted lines. For instance, to recognize P1's first ten spoken programs, we adapted the model with missing line transcripts from the last ten programs of all participants except P1. This process was repeated for each participant.

We examined the impact on recognition accuracy by gradually increasing the amount of adaptation data from 25%, 50%, 75%, and finally, 100% on recog-

nition accuracy. We created adaptation sets by randomly sampling 25%, 50%, and 75% of the missing transcripts combined with 25%, 50%, and 75% of the highlighted transcripts, respectively, and finally, 100% of the transcripts. This random sampling was repeated ten times, and the average accuracy was taken.

4.3.2 Results from A Commercial Recognizer

We utilized simple heuristic rules to post-process the recognition results from Google's recognizer by converting all numbers and symbols into their corresponding words. This was necessary to ensure a fair comparison with our human transcriptions, which also spelled out all numbers and symbols. Using human transcripts, we calculated the Word Error Rate (WER) for the post-processed recognition results. WER was computed by adding up the number of insertions, deletions, and substitutions in the recognition result compared to the reference transcript, dividing by the total number of words in the reference, and then multiplying by 100.

The baseline model had a high WER, with 28.3% for novices and 23.1% for experts (Table 4.1). The highlighted and missing conditions provided similar improvements when adapting the language model, indicating that each condition contains enough variation in spoken code to enhance the model's learning. Gradually increasing the number of transcripts used for adaptation led to a significant reduction in error rates. As illustrated in Figure 4.1, with just 25% of the adaptation transcripts, the WER dropped notably by 18.4% relative to the baseline across all participants. Further increasing the number of transcripts resulted in

Adaptation Data	WER (%)	
	Novice	Expert
None (baseline)	28.3 ± 3.0	23.1 ± 2.0
All missing transcripts	22.8 ± 2.0	18.9 ± 2.0
All highlighted transcripts	23.0 ± 2.0	17.9 ± 2.0
25% missing + 25% highlighted transcripts	23.3 ± 2.0	18.7 ± 2.0
50% missing + 50% highlighted transcripts	22.5 ± 2.0	18.1 ± 2.0
75% missing + 75% highlighted transcripts	21.8 ± 2.0	17.6 ± 2.0
All missing and highlighted transcripts	20.8 ± 2.0	17.0 ± 2.0

Table 4.1: Word error rate (WER) using Google speech recognizer. ± values represent sentence-wise bootstrap 95% confidence intervals calculated by averaging WERs for each sentence across all repetitions.

a slight additional reduction in error rate, although the improvements began to diminish. This suggests that the model captured most of the essential patterns and information early on, making additional data less beneficial. On average, using 100% of the transcripts for adaptation reduced the WER by 27% relative to the baseline across all participants.

Table 4.2 shows some examples of our human reference transcripts, recognition results using the base model from Google, and recognition results using language model adaptation on 100% of the transcripts. Model adaptation improved the recognition of homophones such as "for" versus "four", "i" versus "eye", "u" versus "you" and "two" versus "to". Recognition of "for", "i", "u" and "two" were improved by 70%, 74%, 8%, and 9% relative respectively. This suggests that the model learned the context in which these words were used.

Adapting the language model with all 444 transcripts resulted in an average

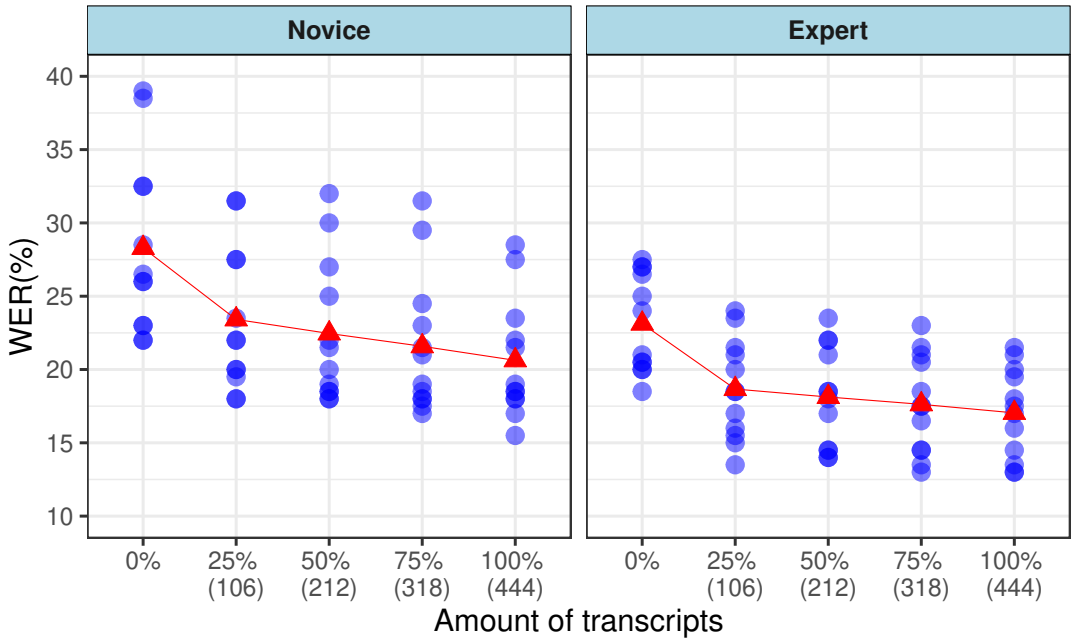


Figure 4.1: Comparison of Word Error Rate (WER) using increasing amounts of adaptation transcripts for the novices (left) and the experts (right). The mean value is marked as a red triangle. The x-axis shows the percentage of transcripts used and the exact number of lines in parentheses.

WER of 19% across participants, indicating that the system inaccurately recognized nearly one out of every five words spoken. This level of performance is suboptimal, especially compared to the low WERs achieved in recent years for natural language speech recognition [4]. Additionally, uncommon words in natural language, such as "int" and "num" were frequently misrecognized, with error rates of 99% and 74%, respectively, even after model adaptation.

Model	Text
human	`string very large string two equals world`
base model	`string very large string to equals world`
adapted model	`string very large string two equals world`
human	`while num greater than equal to one`
base model	`well none greater than equal to one`
adapted model	`while none greater than equal to one`
human	`for i equals one i less than or equal to n i plus plus`
base model	`four equals one i less than or equal to an eye plus twelve`
adapted model	`four equals one i less than or equal to n i plus plus`
human	`large num counter minus minus`
base model	`large and dumb counter minus minus`
adapted model	`large num counter minus minus`

Table 4.2: Recognition results used the base model and the adapted model on 100% of the transcripts. Recognition errors are highlighted in red and underlined.

4.4 Experiments with Research Recognizer

The goal of using a research recognizer was to overcome the limitations of commercial speech recognizers by incorporating more domain-specific audio and text data to improve both the acoustic and language models. We utilized wav2vec 2.0 [4] that can learn speech representations directly from raw audio waveforms without needing a large amount of labeled training data. wav2vec 2.0 is a self-supervised model for speech recognition that processes raw audio waveforms using a multi-layer convolutional neural network [30] to extract key phonetic features, known as latent speech representations. These representations are then fed into a Transformer network, which learns context by predicting masked audio portions based

on the surrounding context. The model's output consists of context-aware representations, which can be fine-tuned for specific tasks such as automatic speech recognition.

In this work, we fine-tuned the model on a dataset of spoken programs to enhance its ability to recognize line-by-line spoken code. This process involved improving the acoustic model by training it with domain-specific audio data and decoding it with a language model trained with domain-specific text data. We investigated the recognition accuracy of models trained with general spoken English versus spoken programming language. To our knowledge, this is the first work focused on recognizing line-by-line spoken programs.

4.4.1 Datasets and Preprocessing

SpokenJava 1.0 Dataset

We use the SpokenJava 1.0 dataset collected in Studies 1 and 2 described in Chapter 3. We split the data into train, development (dev), and test sets with an 80/10/10 ratio. We split data by speakers and target lines of code to measure recognition performance on unseen programs spoken by different users. The dataset was divided into 10/5/5 programming statements, and the corresponding utterances were included in the train/dev/test sets. The relatively compact SpokenJava dataset consisted of 269/27/28 utterances, 29/6/6 users, and 60/5.7/6.9 minutes of audio for the train/dev/test sets, respectively. It includes various programming constructs such as method signatures, if-else statements, loops, input-

output statements, arrays, single and multi-line comments, decrement operations, math statements, and variable declarations. This dataset is unique as it focuses on spoken code, unlike existing datasets that contain written code comments or function descriptions. The transcripts of SpokenJava 1.0 are available online[3].

CodeSearchNet and CodeXGLUE Datasets

In our language modeling experiment, we used a dataset from CodeSearchNet [26], encompassing six programming languages, including Java. The CodeSearchNet corpus comprises 2 million examples from open-source libraries hosted on GitHub. Each data point includes a method, its corresponding method header documentation, and metadata such as repository information. This dataset was later filtered to create the CodeXGLUE [34] dataset. The filtering process involved removing examples where the code could not be parsed into an abstract syntax tree, excluding examples with fewer than three or more than 256 tokens in the documents, discarding documents containing special tokens (e.g.,“ <img>” or “https”), and eliminating non-English documents. An example is shown in Table 4.3.

We then selected all 181K Java examples from the CodeXGLUE dataset and extracted only the docstrings or comments. To better align this dataset with our spoken code dataset, we performed the following preprocessing steps: 1) breaking multi-line descriptions into single lines, 2) splitting camel-case names into separate words, and 3) discarding lines containing symbols such as angle brackets, hyphens, or underscores. This preprocessing resulted in a refined dataset of 495K

[3] https://osf.io/h6nk4

Documentation:

```
/**
 * Sets the value of the given variable.
 * @param name the name of the variable to set
 * @param value the new value for the given variable
 */
```

Code:

```
public void setVariable(String name, Object value) {
    if (variables == null)
        variables = new LinkedHashMap<>();
    variables.put(name, value);
}
```

Table 4.3: Example of a Java method and associated documentation from the CodeXGLUE [34] dataset.

examples. An example of a single-line comment from this dataset is: `"sets the value of the given variable"`. Although this dataset does not consist of spoken programs, comments often describe the purpose and functionality of code in natural language. Using this dataset could help the language model better understand the vocabulary and context related to programming.

4.4.2 Fine-tuning wav2vec2

We conducted fine-tuning experiments with the open-source wav2vec 2.0 model [4], initially trained on natural English speech. Our primary goal was to explore the model's ability to adapt to spoken Java code. Additionally, we aimed to evaluate the impact of data diversity and quantity on the model's performance.

For our experiments, we used the Fairseq toolkit [45] following the settings specified in [4]. We employed a learning rate of 1×10^{-4} with the Adam optimizer, utilizing a tri-state rate schedule to enhance convergence. This schedule included warming up the learning rate for the first 10% of updates, maintaining it constant for the subsequent 40%, and then linearly decaying it. We applied a layer dropout rate of 0.1 to prevent overfitting. The models were fine-tuned using 16-bit precision on two NVIDIA RTX 2080 Ti GPUs with a batch size of 4 samples per GPU. All network parameters were updated except for the feature encoder. We set the time-step and channel mask probabilities to 0.65 and 0.25, respectively, as recommended in [4].

We began with the wav2vec 2.0 base model, pre-trained on 960 hours of unlabeled spoken English data from the Librispeech dataset. This model comprises 12 transformer layers, each featuring 8 attention heads. We fine-tuned this base model using the SpokenJava 1.0 training data for 10K updates, adhering to the configuration in [4]. We observed no significant improvement beyond 10K updates. The number of updates and other hyperparameters were validated using the dev set.

Next, we utilized a fine-tuned checkpoint of wav2vec 2.0, which had undergone training on the LibriSpeech corpus with text labels for up to 300K updates. Building on this checkpoint, we conducted an additional round of fine-tuning using our in-domain SpokenJava 1.0 data. The aim of using a fine-tuned checkpoint was to leverage the knowledge acquired from the labeled natural English data during the initial fine-tuning stage.

4.4.3 Training N-gram Language Models

We trained different n-gram word language models using SRILM [59], testing from trigram to 8-gram. Models larger than 8-gram could not be trained due to insufficient 8-gram occurrences in our SpokenJava dataset.

Our first dataset included human-generated transcripts of individuals speaking lines of Java code. We set aside 5% of this data as a held-out dev set, leaving 255 sentences (4K words) for training. Additionally, we trained separate 4-gram language models on single-line comments from CodexGlue [34] (495K sentences, 4M words) and normalized LibriSpeech (40M sentences, 803M words). All text was uppercased.

Each model employed modified Kneser-Ney smoothing with a vocabulary of 203K words, mapping out-of-vocabulary words to an unknown token. The vocabulary was constructed by merging 200K words from LibriSpeech, all words from our primary training corpus, and words appearing at least five times in CodexGlue. The five-occurrence threshold helped exclude idiosyncratic non-camel-case multi-word combinations.

We evaluated each model based on perplexity, which measures how well a language model predicts a sequence of words. A lower perplexity indicates that the model predicts the next word more effectively in a sequence. Our evaluation revealed that the 4-gram model consistently yielded the best performance across all datasets. Detailed results for all n-gram models are provided in the Appendix B.2.

Subsequently, we created a mixture model by linearly interpolating the 4-gram models from the SpokenJava, CodeXGlue, and LibriSpeech datasets. A mixture model combines multiple language models, each trained on different datasets, to create a single model that leverages the strengths of each component. We optimized the mixture weights to minimize the perplexity on the held-out dev set, ensuring that the resulting model was well-suited to recognizing spoken programming languages.

4.4.4 Beam Search Decoding

We evaluated our fine-tuned models by calculating the Word Error Rate (WER) using beam search decoding with both the LibriSpeech 4-gram model and our best 4-gram mixture model across various wav2vec2 models. We used a lexicon-based beam search decoder available in the Flashlight framework [48] for decoding. The decoder aims to maximize:

$$\log P_{\text{AM}}(\hat{y}|x) + \alpha \log P_{\text{NGRAM}}(\hat{y}) + \beta|\hat{y}| \tag{4.1}$$

where $\hat{y}$ represents the output sequence, x is the input, p_{AM} corresponds to the acoustic score, p_{NGRAM} represents the language model score, and $|\hat{y}|$ is the characters in the transcription (including spaces). α is the language model weight, and β is the word insertion penalty, both of which were optimized based on minimizing WER on the Java dev set. The word insertion penalty helps balance the contribution of the acoustic and language models.

We initially set the beam width to 500 for beam search decoding and tuned other parameters, such as language model weights and word insertion penalties for all models following [4]. Our hyperparameter tuning searched over language model weights in [0, 5] and word insertion penalties in [-5, 5] using Bayesian optimization[4] for 128 trials. The best hyperparameters for each model are included in the Appendix B. Once the optimal parameters were identified, we gradually decreased the beam width down to 5 to find a better balance between speed and accuracy. We found that a beam width of 35 provided the optimal balance, decoding at 11.3 sentences per second and 1004.7 tokens per second. Increasing the beam width to 40 did not improve accuracy and reduced decoding speed to 10.85 sentences per second and 965.6 tokens per second. Larger beam widths continued to reduce performance, with a beam width of 500, dropping to 3.5 sentences per second and 312.8 tokens per second.

4.4.5 Rescoring with a Transformer Language Model

Motivated by the success in enhancing ASR recognition by rescoring with a transformer language model [32], we employ a similar strategy. We utilize the CodeGen-NL model with 350 M parameters from the Hugging Face Transformers Library [68]. CodeGen was trained on the Pile dataset[5], a portion of which includes GitHub code repositories. Using a causal language modeling objective, we fine-tuned the model with our SpokenJava corpus.

[4]https://github.com/bayesian-optimization/
[5]https://pile.eleuther.ai/

We conducted a grid search to optimize hyper-parameters by minimizing per-token perplexity on the held-out dev set. We searched over learning rates in [1e-5, 5e-5], weight decays in [0.001, 0.1], epochs in [1-10], and training batch sizes of 2 and 4. The optimal configuration included a learning rate of 5e-5, batch size of 2, and 3 epochs.

Fine-tuning was performed using two NVIDIA RTX 2080 Ti GPUs, while inference for rescoring was executed on a single GPU.

We rescored the top 50 candidates from our first pass search, calculating a weighted linear combination following [25]. The re-estimated final ranking for each candidate was:

$$\log P_{\text{AM}}(\hat{y}|x) + \alpha_1 \log P_{\text{NGRAM}}(\hat{y}) + \alpha_2 \log P_{\text{NLM}}(\hat{y}) + \beta|\hat{y}| \tag{4.2}$$

Here, $\hat{y}$ represents the output sequence, x is the input, P_{AM} corresponds to the acoustic model score, P_{NGRAM} and P_{NLM} represent the N-gram and transformer language model scores respectively, α_1 and α_2 are the weights assigned to the N-gram and transformer language model scores, respectively, and β is the weight assigned to the length penalty term $|\hat{y}|$.

Our hyperparameter tuning for rescoring searched over α_1 in [0.0, 1.0], α_2 in [0.0, 1.0], and β in [-5.0, 5.0] using Bayesian optimization[6] for 128 trials. The best parameters identified were $\alpha_1 = 0.3$, $\alpha_2 = 0.2$, and $\beta = 0.3$.

[6]https://github.com/bayesian-optimization/

Labeled Data	Dev	Test
LibriSpeech (960h)	31.1	28.4
SpokenJava (1h)	18.0	25.9
LibriSpeech (960h) + SpokenJava (1h)	**8.1**	**8.7**

Table 4.4: Word Error Rate (WER) on the dev and test sets varying the labeled data used for fine-tuning. Results obtained by greedy decoding without a language model.

4.4.6 Results from wav2vec 2.0

We calculate WER for all models on our dev and test sets in three scenarios: without external language model decoding, with beam search decoding using an N-gram language model, and with beam search decoding followed by rescoring with a transformer language model. For all evaluations, we selected the model checkpoint with the lowest WER on the dev set.

The pre-trained wav2vec 2.0 base model, which had no prior exposure to spoken programs and was initially fine-tuned on 960 hours of labeled LibriSpeech data, showed a high WER of 31.1% on the dev set and 28.4% on the test set, as shown in Table 4.4. With just one hour of SpokenJava, we observed a 42.1% relative improvement in WER on the dev set but only an 8.8% relative improvement on the test set, both compared to the base model (Table 4.4). The model struggled to generalize on the test set, likely due to overfitting from the limited training data and the more diverse programming statements in the test set. However, when we took the model already fine-tuned on 960 hours of labeled LibriSpeech data and further adapted it with spoken programs, it achieved a WER of 8.1% on the dev set and 8.7% on the test set. It could be that adding extensive natural En-

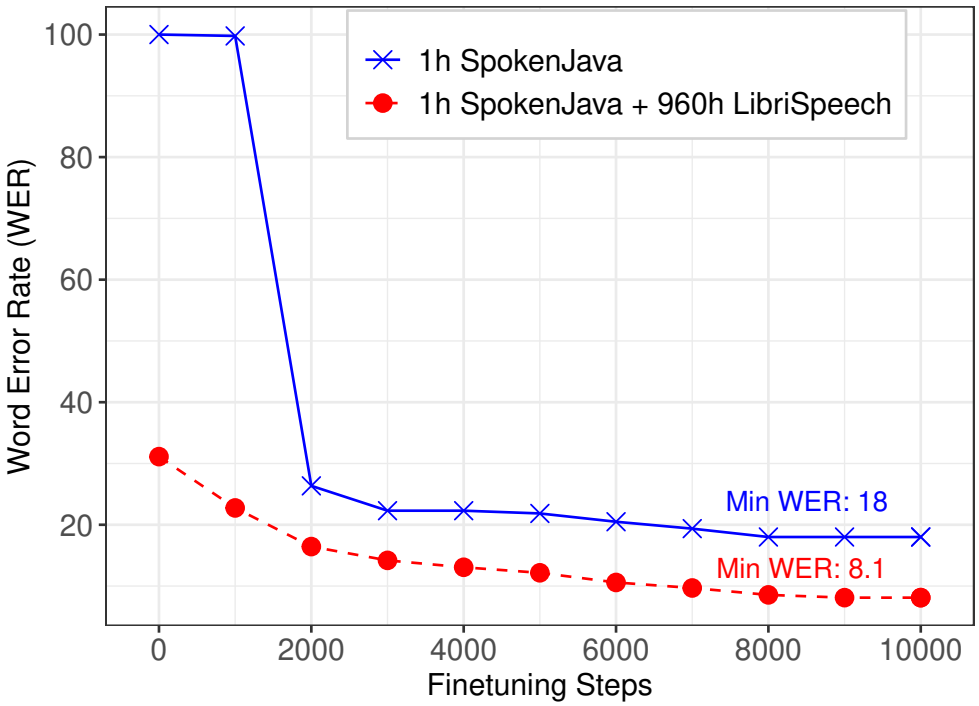


Figure 4.2: Word Error Rate (WER) on the dev set for different models for different fine-tuning steps. The minimum WER values for each model are highlighted.

glish data improved the model's capability to handle diverse speakers and spoken content.

In our N-gram language modeling experiment, we found that a mixture model produced the best results. Our best N-gram model was a combination of 4-gram SpokenJava (weighted at 0.7), 4-gram LibriSpeech (weighted at 0.2), and 4-gram CodexGlue (weighted at 0.1). We think this improvement can be attributed to the inherent natural English language in spoken Java programs, which benefited from LibriSpeech data, and learning written comments from a partially in-domain CodexGlue dataset.

Model	LM	Dev	Test
LibriSpeech (960h)	4-gram-libri	22.3	24.1
LibriSpeech (960h)	4-gram-mixture	11.2	11.3
LibriSpeech (960h) + SpokenJava (1h)	4-gram-libri	6.5	7.6
LibriSpeech (960h) + SpokenJava (1h)	4-gram-mixture	5.6	6.0
LibriSpeech (960h) + SpokenJava (1h)	4-gram-mixture + rescoring	**4.5**	**5.5**

Table 4.5: Word Error Rate (WER) on the SpokenJava dev and test sets of different wav2vec and language model combinations.

Table 4.5 shows the impact of different language models on the fine-tuned models, with the 4-gram LibriSpeech model as the baseline. Despite having a limited amount of in-domain text data, we substantially improved the WER decoding with our 4-gram mixture model. Specifically, for the 960h-Libri fine-tuned model, the WER decreased by 49.8% and 53.1% relative. Using the 4-gram mixture model with one hour of SpokenJava data, the WER decreased by 13.8% and 21.0% on dev and test sets, respectively, relative to the LibriSpeech 4-gram model with one hour of SpokenJava data. Rescoring with our adapted transformer model further improved accuracy, resulting in a 19.6% and 8.3% relative reduction in WER on the dev and test sets, respectively, compared to the best 4-gram decoded model.

In our experiment with a commercial speech recognizer, we tested performance using Google's speech-to-text on our collected data, which resulted in a high WER of 25%. Even with language model adaptation, the WER remained high at 19%. In contrast, our best-trained wav2vec 2.0 model achieved a significantly lower WER of 5.5% on the test data. Although the test data in the experiments with

Model	Text
Human	items at index i is equal to scan dot next int
Base model	items a index eyes equal to scan dot next int
Best adapted model	items at index is equal to scan dot next int
Human	constructor public employee int age comma double salary
Base model	instructor public employ n comet double salary
Best adapted model	instructor public employ int age comma double salary
Human	for int i equal zero i less than five i plus plus
Base model	or in equal zero eye less than five eye plus plus
Best adapted model	for int i equals zero i less than five i plus plus
Human	create a public static method called print phrase that takes two arguments the first is a string phrase and the second is a double called num
Base model	create a public setoc method called print phrase that takes two arguments the first is a string phrase and the second is a double called numb
Best adapted model	create a public static method called print phrase that takes two arguments the first is a string phrase and the second is a double called num

Table 4.6: Example human transcripts and predictions from the 960h-libri model decoded with a 4-gram LibriSpeech language model (base model) and our rescored model. Word errors are underlined and in red.

Google's speech-to-text and wav2vec were two different subsets of our SpokenJava 1.0 dataset, they were similar in nature.

We observed the predictions from the base model and our best-trained model(Table 4.6). The base model with knowledge of natural English learned the natural description of a small method but struggled with some fundamental Java keywords

like `"static"` and abbreviated terms like "num". Our best model with knowledge of both natural English and spoken Java programs learned some Java terms like "int", "for" and some common variable names like "i" or "num". This suggests effective domain-specific learning. Our best model still struggled with text containing a blend of natural language words like "employee" and code-related words like "public" or "int" indicating potential issues arising from sparsely labeled data.

4.5 Discussion and Limitations

Our study aimed to enhance the accuracy of speech recognition systems for spoken Java programs. Initially, we utilized Google's speech recognizer, which revealed significant difficulties in accurately recognizing programming language. Although this general-purpose recognizer could be adapted to some extent for specific tasks, there were limitations. Despite a 26.4% relative reduction in WER from the baseline with our transcribed spoken code, the system's performance plateaued with further data adaptation. This plateau suggests that the underlying models were not fully equipped to handle the unique vocabulary and syntax of programming languages. This might be due to the specific limitations of Google's language model adaptation algorithm in handling the unique syntax and vocabulary of programming languages, as well as potential challenges in adapting to the diverse accents and speaking styles specific to programming speech

Given these limitations, we turned to the wav2vec 2.0 research recognizer, allowing precise model customization. By fine-tuning both the acoustic and lan-

guage models and incorporating transformer-based rescoring, we achieved substantial improvements in accuracy. This underscores the importance of having control over model components to address the specific needs of spoken programming language recognition effectively. From our experiment with the research recognizer, we observed marked improvements in accuracy by fine-tuning the wav2vec 2.0 model with both general English speech data (labeled 960 hours of LibriSpeech) and a small amount of domain-specific speech data (labeled one hour of Spoken-Java). This exposure of both data enabled the model to better understand the unique vocabulary and syntax of programming languages. Our results showed fewer errors and more accurate transcriptions of key programming terms, indicating that the model effectively adapted to the spoken programs.

However, our best-trained wav2vec 2.0 model sometimes struggled with certain terms and symbols unique to programming, such as “int” and “num” which were often misrecognized. One potential solution could be data augmentation. Data augmentation techniques, such as generating text-to-speech audio from a large dataset of comments, could provide additional training data and enhance model performance. A strong language model can significantly improve the accuracy of a speech recognizer, which requires a large amount of text data. This can be achieved by generating synthetic text data through few-shot learning, which involves training a large language model with a limited amount of annotated data. However, as an initial exploration, our study establishes the groundwork for future research in this domain.

4.6 Conclusions

In conclusion, our study lays a solid groundwork for future research in the field of spoken programming language recognition. The average WER achieved by Google's speech-to-text system, combining results from both novice and expert programmers, was 18.9%. This provided initial insights and informed our data collection strategy, ensuring adequate capture of data variations. By subsequently fine-tuning the wav2vec 2.0 with a mix of general speech data and in-domain data, we achieved a low final error rate of 5.5% on the test set, demonstrating the effectiveness of this domain-specific adaptation. These findings emphasize the potential of adapted speech recognition models to greatly improve the accuracy of recognizing spoken programming languages. Chapter 5 builds on this foundation, exploring how to convert recognized spoken words into the target line of code.

Chapter 5

Text-to-Code Translation

5.1 Introduction

Generating code poses more challenges than standard natural language generation due to its strict syntax and semantic rules. Unlike natural language, where minor grammatical errors may still convey the intended meaning, even a minor error in code, such as a missing semicolon or a misplaced dot, can drastically alter the code's functionality or make it entirely incorrect.

In the previous chapter, we detailed the first step of our pipeline: recognizing line-by-line spoken programs using our fine-tuned wav2vec model. In this chapter, we describe the second step of our pipeline: generating single-line code from the transcript of a spoken single-line Java program. We detail our machine translation experiments to convert a single-line text to a single-line code.

5.2 Related Work

In recent years, there has been significant interest in automating software engineering tasks to enhance programmers' productivity. These tasks include text-to-code, code-to-text generation, code summarization, code translation between programming languages, bug detection, and code completion [34, 18, 60, 36, 52]. One primary task in this domain is generating a code block from comments or docstrings. A docstring is a type of comment used to describe the purpose and functionality of a function or module in the code.

Early approaches, such as the sequence-to-sequence model for code generation introduced by Ling et al. [33], laid the groundwork by demonstrating how natural language descriptions could be converted into code. Building on this, Yin et al. [69] improved upon traditional decoders by integrating grammatical rules and generating abstract syntax trees before converting them into code. They employed a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) units. RNNs are designed to handle sequential data by tracking previous steps to influence future ones. However, they struggle with long-term dependencies, where critical information needed for predictions is far apart, making it difficult for RNNs to retain information over long sequences [24, 8]. In addition, RNNs can not be efficiently trained on large datasets as they require sequential processing of tokens, making parallelization difficult since each token's computation depends on the previous hidden state.

The development of the transformer architecture [64] and the availability of ex-

tensive codebases [26, 34, 12, 11] have marked a new era of advancements in code generation. Transformer models process entire sequences simultaneously, offering a significant advantage over RNNs by capturing long-range token dependencies. This advantage primarily comes from the self-attention mechanism, which assesses the relevance of each token in the input sequence by considering the entire context when predicting each token in the output. This makes them particularly effective for tasks like code generation, where understanding the broader context and maintaining long-term dependencies are required.

OpenAI's **GPT** family models [70, 9] have demonstrated the capability to generate code based on prompts. These models are capable of learning from minimal examples with in-domain data. GPT-3, for instance, showcased the feasibility of few-shot learning, where the model is given only a few examples to learn from. This achieves competitive results in various tasks, such as code generation. One such model is CodeGPT [34], a transformer-based language model pre-trained on programming languages. CodeGPT supports text-to-code generation tasks. It's based on the architecture and training objectives of GPT-2 [70]. CodeGPT is pre-trained on Python and Java code from the CodeSearchNet [26] dataset. This dataset includes pairs of functions and code snippets extracted from GitHub, with 1.1 million Python and 1.6 million Java code snippets.

CodeBERT [18] is a pre-trained model that supports tasks like code documentation generation and is based on Google's BERT [16] model. BERT is a masked language model that predicts the missing words in a sentence based on the context. The training data for CodeBERT comes from the CodeSearchNet

[26] dataset, which includes bimodal (a code snippet paired with its documentation) and unimodal (a code snippet) data points. The dataset contains 2.1 million bimodal and 6.4 million unimodal data points across six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go.

PLBART [1] is a model that can summarize and generate code from text. It is pre-trained on a large collection of Java and Python functions and associated comments. The training data for PLBART consists of 470 million functions for Java and 210 million functions for Python all extracted from GitHub as well as 47 million questions and answers from StackOverflow.

Meta's Code Llama [53] is a set of large language models designed for programming tasks based on the Llama 2 [62] architecture. These models achieve state-of-the-art performance and can complete code, handle long input contexts, and follow prompts to generate code. Code Llama supports multiple programming languages, including Python, Java, and C++. The models come in different sizes: 7, 13, and 34 billion parameters, each trained on 500 billion tokens, and a larger 70B model trained on 1 trillion tokens. The training data consists of a collection of publicly available source code. Additionally, 8% of the training samples are from natural language datasets related to code, including discussions about code and code snippets in question-and-answer forums.

Gemini [55], released by Google's DeepMind, is a multimodal large language model. Gemini can complete missing code and generate code based on prompts, primarily producing code snippets. It is trained on a diverse dataset that includes data from web documents, books, and code, as well as image, audio, and video

data or internally generated from an internal dataset.

All these models excel at understanding code, but they are capable of generating code snippets from the method header comments. However, our task focuses on generating a single line of code from a line of spoken programming transcript rather than from comments. This distinction highlights the unique challenge of our work, as natural speech often lacks the structured context provided by the written comments.

5.3 Text-to-Code Translation

5.3.1 Dataset and Preprocessing

We utilized the SpokenJava 2.0 dataset[1] from Study 3, as described in Chapter 3. This dataset provided a diverse set of programming statements, including spoken code transcripts for single-line code, the corresponding target code, and the target line's preceding and following code snippets. Incorporating these contextual snippets allowed us to investigate whether providing additional context could improve the model's ability to generate accurate code.

The dataset was split into training, development (dev), and test sets, with the training set consisting of 833 lines, the dev set having 115 lines, and the test set containing 295 lines. We split the dataset by participants, with 67.9% of participants' data used for training, 10.7% for development, and 21.4% for testing. Splitting by participants ensured that each split contained all programming con-

[1] https://osf.io/6axd2/?view_only=518e06e29e1e4e0b90a9447e6e56c84f

structs we used and that data from any participant was included in only one of the three sets. Thus eliminating any potential for data leakage and ensuring the independence of the evaluation sets.

5.3.2 Evaluation Metrics

In this work, we used exact match accuracy and CodeBLEU [34] to evaluate our adapted models. Exact match accuracy was calculated by comparing the generated line of code with the reference line of code and measuring the percentage of cases that matched exactly. This metric provides a straightforward assessment of correctness but does not account for partial correctness or the syntactic and semantic quality of the generated code.

For CodeBLEU, we used a composite metric that combines the scores of four sub-metrics to evaluate the quality of generated code. Each sub-metric was calculated, and their weighted average, with a standard weight of 0.25 for each sub-metric, was taken to get the final CodeBLEU score.

The quality of machine translation models is typically assessed by the BLEU (Bilingual Evaluation Understudy) score [47]. While BLEU was originally developed for evaluating machine translation of natural languages, it has limitations in evaluating code due to its focus on natural language, neglecting syntactic and semantic features.

CodeBLEU [50] is used to evaluate the quality of generated code for tasks such as code generation and translation. Research [50, 17, 34] indicates that CodeBLEU aligns more closely with human evaluations than BLEU or exact match accuracy

for evaluating code generation models. CodeBLEU is a composite metric that combines the scores of four sub-metrics:

- **Standard BLEU**: Measures n-gram overlap, treating all code tokens equally.
- **Weighted BLEU**: Assigns higher importance to keywords. For example, in Python code, keywords such as `"def"`, `"return"`, and `"import"` are given more weight.
- **Abstract syntax tree**: Analyzes the code's syntactic structure.
- **Data-flow graph**: Examines the semantic flow and variable dependencies.

The final CodeBLEU score is a weighted average of these sub-metrics, with a standard weight of 0.25 giving the best results [50].

5.3.3 Fine Tuning Code Llama

Given the significant computational resources required for training and deploying large-scale LLMs, we focused on code-targeted LLMs and chose Code Llama 7B. Code Llama is capable of generating code and understanding the natural language of code, making it suitable for our task. Additionally, Code Llama's capability to handle long input contexts is particularly advantageous for our task. Due to resource limitations, we fine-tuned Code Llama 7B using 4-bit quantization [13]. The 4-bit quantization technique reduces model weights to 4 bits instead of the standard 16 or 32 bits. This allowed us to train the Code Llama 7B model on available hardware.

Prompt with no context

```
Convert the given text into a single line of Java code.

### Text:
add value to total value

### Output:
totalValue += value;
```

Prompt with pre-context

```
You are given a text and a preceding
code snippet that comes before the output. Convert the text into a single line
of Java code.

### Text:
if is eligible

### Preceding Code:
boolean isEligible = checkEligibility(user);

### Output:
if (isEligible)
```

Prompt with preceding and following code context

```
You are given a text, a preceding code snippet that comes before the output, and
a following code snippet that comes after the output. Convert the text into a
single line of Java code.

### Text:
if is eligible

### Preceding Code:
boolean isEligible = checkEligibility(user);

### Following Code:
processResult(result, isEligible);

### Output:
if (isEligible)
```

Table 5.1: Sample prompts used for fine-tuning Code Llama

Code Llama uses a prompt, an instruction given to the model, to help the model understand a task. A well-crafted prompt can significantly enhance the model's performance by clearly defining the task. We experimented with various prompts and evaluated the model on the dev set based on the CodeBLEU score, selecting the best-resulting prompt.

We conducted three sets of experiments to evaluate the model's performance:

1. **First experiment - without context**: In this baseline experiment, we fine-tuned the model using only the input text and target code as labels in the prompt without including any surrounding code snippets. This helped us understand the model's performance when generating code solely from the spoken transcript. An example prompt is given to the model for training, which includes the instruction, human transcript of the spoken single-line Java code as text, and the target line of code as the output (Table5.1).

2. **Second experiment - with preceding context:** In the second experiment, we included the preceding code context to provide the model with additional information about the code that came before the target line as shown in Table5.1. This scenario mirrors real-world programming situations where developers add a new line to existing code. Due to memory constraints, only up to two lines of preceding code were included. If any lines contained only punctuation (e.g., only a closing curly brace) without code content, we continued reading subsequent lines until we encountered lines with code content.

3. **Third experiment - with both preceding and following context:** In the third experiment, we included both preceding and following code snippets to give the model context from both directions as presented in Table5.1. This approach aimed to see if additional context before and after the target line of code would further improve the model's performance. This scenario mirrors real-world programming, where developers often need to modify a line in the middle of the code. We experimented with two preceding and two following code lines that could fit the memory.

We conducted a grid search for each experiment to optimize hyperparameters based on the CodeBLEU score on the dev set. The hyperparameter ranges included a learning rate between [1e-5, 3e-3], weight decay between ([1e-3, 1e-1]), and batch size between ([4, 8]). We conducted a grid search for each experiment to optimize hyperparameters based on the CodeBLEU score on the development set. The hyperparameter ranges included a learning rate between [1e-5, 3e-3], weight decay between [1e-3, 1e-1], and batch size between [4, 8]. We used gradient accumulation with an accumulation step of two to manage memory constraints. Gradient accumulation simulates a larger batch size by accumulating gradients from multiple small batches before performing an optimization step. This approach allowed us to effectively increase the batch size without exceeding memory limits.

We ran the training over five epochs, employing early stopping to prevent overfitting. Early stopping stops the training process when the model's performance on the validation set does not improve further. We utilized a patience of three, which

means training was terminated if there was no improvement for three consecutive epochs. The best checkpoint from the training process was used to evaluate the model on the test set. All fine-tuning experiments were performed using 16-bit precision to reduce memory consumption and computational requirements. The experiments were executed on two NVIDIA RTX 2080 Ti GPUs.

We incrementally added data to our best-performing model to determine the optimal training data needed. We started with 25% of the data, then increased to 50%, 75%, and finally 100%. We shuffled and randomly selected these data portions using three random seeds and then averaged the resulting CodeBLEU scores.

Finally, with the 100% SpokenJava 2.0 training data, we added SpokenJava 1.0 training data (269 lines) used in the wav2vec fine-tuning experiment presented in Chapter 4. This resulted in a total of 1102 lines. We have added the before and after context for each example in the SpokenJava 1.0 dataset from the code snippets we used in the data collection studies. Our goal was to determine whether increasing the amount and variability of training data would improve the model's ability to handle the nuances of natural language when translating spoken code into actual code.

5.3.4 Inference

We selected the best model from each experiment and performed inference using the same prompt format the model was trained with, as shown in Table 5.1. For this process, we employed a single NVIDIA RTX 2080 Ti GPU with a batch size

of eight, the maximum that could fit into memory.

We used beam search decoding, which tracks multiple hypotheses to select the most probable sequence. However, Beam Search was computationally expensive, and we could only fit up to beam three with a batch size of eight.

In addition, we evaluated the performance of our best-adapted model on the recognized transcripts from our fine-tuned wav2vec model described in Chapter 4. Recognized transcripts might contain errors due to recognition inaccuracies, unlike human transcripts. Our goal was to assess the model's capability to manage real-world scenarios where input data from a speech recognizer may contain inaccuracies.

5.4 Results

This section presents the CodeBLEU and exact match accuracies for different models evaluated on the dev and test sets. The models include the base model, the fine-tuned no-context model with no context, the fine-tuned model with preceding context, and the fine-tuned model with both preceding and following context. We selected the model checkpoint with the highest CodeBLEU score on the dev set for all evaluations.

5.4.1 Model Performance Across Different Contexts

The base model achieved a CodeBLEU score of 52.9% on the dev set and 56.9% on the test set. As shown in Table 5.2, transitioning from the base model to the fine-

Model	Pre-context	Post-context	Dev		Test	
			CB(%)	EM(%)	CB(%)	EM(%)
Code Llama-7B	–	–	52.9	19.8	56.9	23.4
Code Llama-7B-adapted	–	–	64.1	37.1	75.9	47.1
Code Llama-7B-adapted	2	–	69.7	47.4	81.5	58.0
Code Llama-7B-adapted	2	2	70.4	52.6	83.3	65.8

Table 5.2: CodeBLEU (CB) score and Exact Match (EM) accuracies on dev and test sets for different models trained with the SpokenJava 2.0 dataset.

tuned model without context led to a relative improvement of 21.2% on the dev set and 33.4% on the test set in CodeBLEU score. This substantial gain suggests that fine-tuning the model, even without additional contextual information, enhanced its ability to generate accurate code, likely due to its increased exposure to relevant data during the fine-tuning process. Adding two lines of preceding context further improved performance, achieving a relative CodeBLEU increase of 8.7% on the dev set and 7.4% on the test set. This indicates that even limited contextual information before the target code can substantially improve the model's accuracy. Including two lines of preceding and two lines of following context resulted in minimal gains on the dev set 1.0% and 2.2% on the test set. The smaller relative improvement from adding the following context may be because the model has already captured most of the necessary contextual information from the preceding lines.

Precision in generating line-by-line code is essential because each line must be correct for the entire code to function properly. The base model had exact match accuracies of 19.8% on the dev set and 23.4% on the test set. Fine-tuning without context improved the accuracy remarkably by a relative 87.4% on the dev set

and a relative 101.3% on the test set, highlighting the significant impact of fine-tuning. Adding the preceding context further improved exact match accuracies by a relative 27.8% on the dev set and 23.1% on the test set. Including both preceding and following context led to additional gains, with improvements of relative 11.0% on the dev set and 13.4% on the test set.

5.4.2 Model Performance on Different Constructs

We observed how improvements varied for each programming construct (Figure 5.1). Our dataset had several programming constructs: loops, comments, variable declarations and initialization, method signatures and method calls, mathematical statements, and ifElse statements. We counted the total number of samples for each type of construct and how many were exactly correct in the test set.

The base model correctly generated 29.1% variable declaration and initialization statements as shown in Figure 5.1. Fine-tuning without context significantly enhanced exact match accuracy to 75.5%. When the preceding context was added, the exact match accuracy increased further to 78.2%. Incorporating preceding and following contexts resulted in an exact match accuracy of 85.4%. The base model struggled to translate compound variable names. For example, it produced `“char p = a; char s = s; char w = d; char o = o;”` for the reference code `“char passwd1;”` when translating the spelled-out variable name `“char p a s s w d one”` (Table 5.3). The no-context model also struggled with aggregating variable names, translating this to `“char pA sSw dOne”`. The addition of surrounding context significantly enhanced the model's performance. While the

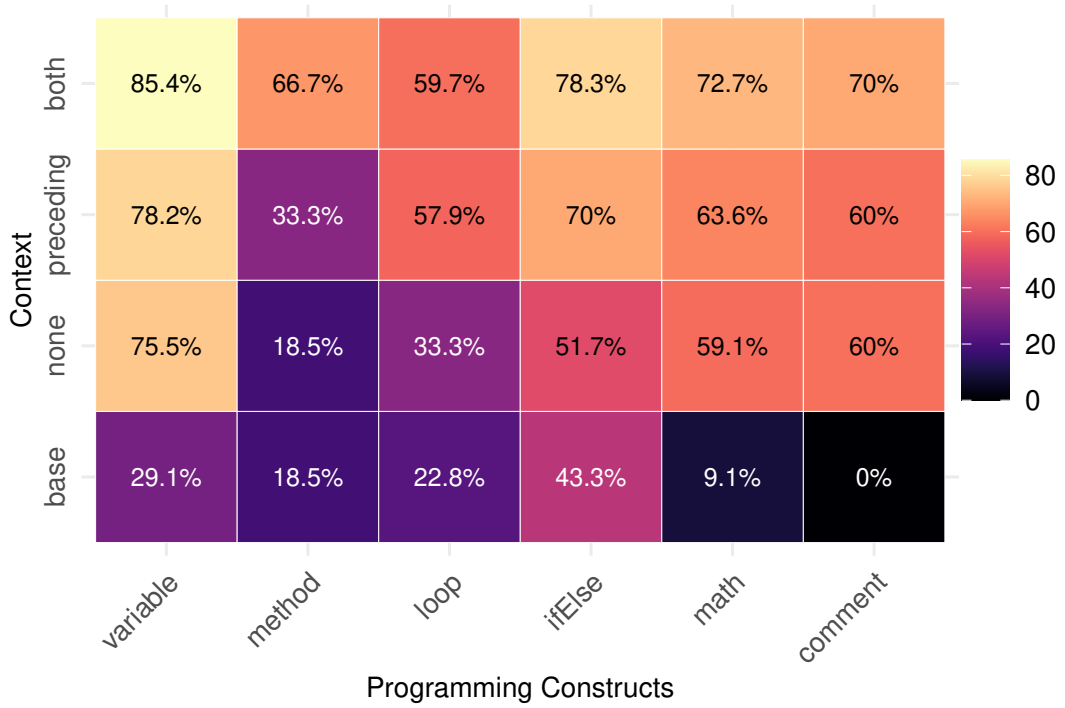


Figure 5.1: Heatmap showing the exact match accuracies for various programming constructs in the SpokenJava 2.0 test set, comparing base and adapted no-context, pre-context, pre- and post-context models.

variable `"passwd1"` wasn't directly mentioned in the preceding context, the target line appeared within a method named `"handlePassword"`. This likely aided the model in inferring the intended variable name. The best-performing model, which incorporated both preceding and following context, accurately predicted `"char passwd1;"`. The repeated mentions of `"passwd1"` in the following context reinforced its correct usage and initialization.

For method signature and method call, the base model had an exact match accuracy of 18.5%. Fine-tuning without context did not change this accuracy.

However, adding the preceding context increased the accuracy to 33.3%. Preceding context alone did not help much for method signatures as it was the very first line with no preceding context in some cases, which explains the significant improvement when the following context was provided.

The base model demonstrated limited accuracy in loop constructs, achieving an exact match accuracy of 22.8%. Fine-tuning without context yielded an improvement, raising the accuracy to 33.3%. Incorporating the preceding context significantly enhanced the performance, increasing the accuracy to 57.9%. The model that utilized both preceding and following contexts showed a further small improvement, achieving an accuracy of 59.7%. We observed that most inconsistencies arose from incorrect variable names in loop statements, even though the overall loop structure was generated correctly.

The base model was better (43.3%) at understanding if-else statements than other constructs. Fine-tuning without context raised this accuracy to 51.7%. The inclusion of the preceding context and surrounding contexts significantly enhanced the accuracy to 70.0% and 78.3%, respectively.

The proportion of correct mathematical operations saw a remarkable increase from the base model accuracy of 9.1% to the accuracy of 59.1% with the no-context model. Adding the preceding context increased the accuracy to 63.6%. Including both contexts further improved the accuracy to 72.7%. This improvement was mostly due to better handling of parentheses and structural differences. For example, the target line `“float x = x1 * (dx+i);”` was correctly translated by all fine-tuned models from the text `“float x equals x one plus d x time i”` but

not by the base model, which missed the parentheses and this changed the meaning of the code. This likely occurred as parentheses were not explicitly mentioned in the spoken transcript.

Fine-tuning without context improved comment generation accuracy to 60%. Adding the preceding context did not change this accuracy, but incorporating both contexts increased it slightly to 70%. This might be because programmers have a tendency to write a comment followed by related code. When the model saw the following context, it could better generate the comment accurately. While generating comments, the base model often incorrectly produced the literal word `"comment"` instead of adding `"//"` or `"/* */"` (Table: 5.3). The base model was originally trained with comment-code pairs and could translate comments to code, but it struggled with understanding how a particular comment line was spoken. The model also struggled to generate comments when the spoken text closely resembled executable code. For instance, the transcript `"comment loop through each buffer of the input array"` had the target code `"// loop through each buffer of the input array"`. The base model and the model trained without context generated an entirely incorrect line `"for(int i=0; i<input.length; i++)"`, misinterpreting the transcript as executable code rather than a comment. The model that incorporated only the preceding context and the model with the surrounding context correctly generated the comment. This improvement can be attributed to the preceding context, which included lines like `"// create the buffer array to return"`, likely helping the model infer that the transcript was meant to be a comment.

Context	Text
target code	`float x = x1 * (dx + i);`
human transcript	`float x equals x one times d x plus i`
base	`float x = x * d + i;`
no context	`float x = x1 * dx + i;`
pre-context	`float x = x1 * dx + i;`
pre & post-context	`float x = x1 * (dx + i);`
target code	`for (int i = 0; i < numToRun; i++)`
human transcript	`for i from zero to num to run increment i`
base	`for (int i = 0; i < num; i++)`
no context	`for (int i = 0; i < num; i++)`
pre-context	`for (int i = 0; i < numToRun; ++i)`
pre & post-context	`for (int i = 0; i < numToRun; i++)`
target code	`char passwd1;`
human transcript	`char p a s s w d one`
base	`char p = a; char s = s; char w = d; char o = o;`
no context	`char pA sSw dOne`
pre-context	`char passwdOne;`
pre & post-context	`char passwd1;`
target code	`setTargetHost();`
human transcript	`call set target host`
base	`setTargetHost(call);`
no context	`call.setTargetHost();`
pre-context	`setTargetHost(newHost);`
pre & post-context	`setTargetHost();`
target code	`// loop through each buffer of the input array`
human transcript	`comment loop through each buffer of the input array`
base	`for(int i = 0; i < input.length; i++)`
no context	`// for (int i = 0; i < input.length; i++)`
pre-context	`// loop through each buffer of the input array`
pre- and post-context	`// loop through each buffer of the input array`

Table 5.3: Example code translations on the SpokenJava 2.0 test set, comparing different models: base model, no context model, model with pre-context, and model with both pre- and post-context.

Model	Human text		Recognized text	
	CB (%)	EM (%)	CB (%)	EM (%)
Code Llama-7b	56.9	23.4	53.7	16.6
Code Llama-7b-adapted	83.3	65.8	76.7	50.2

Table 5.4: Comparison of CodeBLEU (CB) scores and Exact Match (EM) accuracies on the test set between the base model (Code Llama-7b) and the adapted model with pre- and post-code (Code Llama-7b-adapted), using human transcripts and recognized transcripts.

5.4.3 Model Performance on Recognized Speech

We evaluated our test data using our best-fine-tuned wav2vec model, which yielded a recognized transcript with a Word Error Rate (WER) of 10.4%, indicating that roughly 10 out of every 100 words were incorrect. We then used our best-adapted model to translate these recognized single-line transcripts into actual lines of code. Our objective was to determine how well the text-to-code model can understand misrecognized words introduced by the speech recognizer, ultimately producing accurate code.

As shown in Table 5.4, the base model achieved a CodeBLEU score of 53.7% and an exact match accuracy of 16.6%, while our best-adapted model with both preceding and following contexts achieved a higher CodeBLEU score of 76.7% and an exact match accuracy of 50.2%. When translating human transcripts to code, the best-adapted model had a higher CodeBLEU score of 83.3% and an exact match accuracy of 65.8% compared to translating recognized transcripts. This discrepancy is likely due to misrecognition in the predictions from the speech recognizer.

We observed how well the model translated misrecognized text into accurate code. There were 42 instances where our best-adapted model correctly translated the code despite errors in the recognized transcripts from the speech recognizer out of a total of 146 instances with erroneous recognition. A few examples are provided in Table 5.5.

For instance, consider the recognized transcript `"if roles aloud is not equal to null"` with corresponding human transcript `"if roles allowed is not equal to null"`. The base model produced an erroneous code `"if (rolls.aloud != null)"`. In contrast, the best-adapted model correctly generated `"if (rolesAllowed != null)"`. The context provided around the code, specifically the frequent occurrence and use of the term `"allowed"` in the preceding code snippet, helped the fine-tuned model correctly understand and interpret the misrecognized word `"aloud"` as `"allowed"`. This demonstrates the model's capability to leverage the surrounding context to make accurate corrections even from misrecognized words.

5.4.4 Impact of Training Data Size

We evaluated the performance of our best model with preceding and following code contexts across varying amounts of training data: 25%, 50%, 75%, and 100% (Table 5.6). As the training data increases, CodeBLEU and exact match accuracies improve on both the dev and test sets, as shown in Figure 5.2.

Initially, from 25% to 50% training data, the CodeBLEU score on the dev set increased by 7.3% relative and on the test set by 7.0% relative. The Exact Match accuracy on the dev set increased by 9.7% relative and on the test set by 2.6%

Human Transcript	`for template type key in keys`
Recognized Transcript	`for temple type keys in keys`
Target Code	`for (TemplateType key : keys)`
Translated Code	`for (TemplateType key : keys)`
Human Transcript	`plus plus s e q n r`
Recognized Transcript	`plus plus s e que n r`
Target Code	`seqNr++`
Translated Code	`++seqNr;`
Human Transcript	`if roles allowed is not equal to null`
Recognized Transcript	`if rolls aloud is not equal to null`
Target Code	`if (rolesAllowed != null)`
Translated Code	`if (rolesAllowed != null)`
Human Transcript	`boolean login equals true`
Recognized Transcript	`bullion login equals true`
Target Code	`boolean login = true;`
Translated Code	`boolean login = true;`

Table 5.5: Examples of code translations from recognized wav2vec transcripts on the SpokenJava 2.0 test set.

relative. As we move from 50% to 75% training data, the improvements continue but at a slower pace. The CodeBLEU score on the dev set increased by 1.9% relative and on the test set by 2.7% relative. The exact match accuracy on the dev set increased by 5.4% relative and on the test set by 2.7% relative. These continued improvements, albeit smaller, indicate that the model still gains from the additional data. However, the diminishing returns suggest that the model has already learned a substantial portion of the patterns present in the dataset.

From 75% to 100% training data, the improvements further diminish. The

Training Data	Dev		Test	
	CB	EM	CB	EM
25% SpokenJava 2.0	63.3	42.1	73.9	60.9
50% SpokenJava 2.0	67.9	46.2	79.1	62.5
75% SpokenJava 2.0	69.2	48.7	81.2	64.2
100% SpokenJava 2.0	70.4	52.6	83.3	65.8
100% SpokenJava 2.0 + 100% SpokenJava 1.0	74.4	54.3	84.6	69.2

Table 5.6: CodeBLEU (CB) scores and Exact Match (EM) accuracies of Code Llama-7b-adapted with two lines of pre- and post-code context using varied training data amounts. The result is an average of three random choices for 25-75%. SpokenJava 1.0 represents data from Studies 1 and 2, while SpokenJava 2.0 represents data from Study 3 (Chapter 3).

CodeBLEU score on the dev set increased by 1.7% relative and on the test set by 2.6% relative. The exact match accuracy on the dev set increased by 8.0% relative and on the test set by 2.5% relative. The additional examples provide incremental improvements, but the model might have absorbed most of the variations it can learn from the dataset.

Augmenting 100% of SpokenJava 1.0 data (269 lines) collected in Study 1 and Study 2 with 100% of SpokenJava 2.0 led to a CodeBLEU score increase by 5.7% on the dev set and 1.6% on the test set, relative to using 100% of the SpokenJava 2.0 dataset. The exact match accuracies increased by a relative 3.2% on the dev set and 5.2% on the test set. Our SpokenJava 1.0 dataset had the same set of 20 different programming statements spoken by various programmers. With this limited variation in programming statements, the model likely did not encounter new types of code.

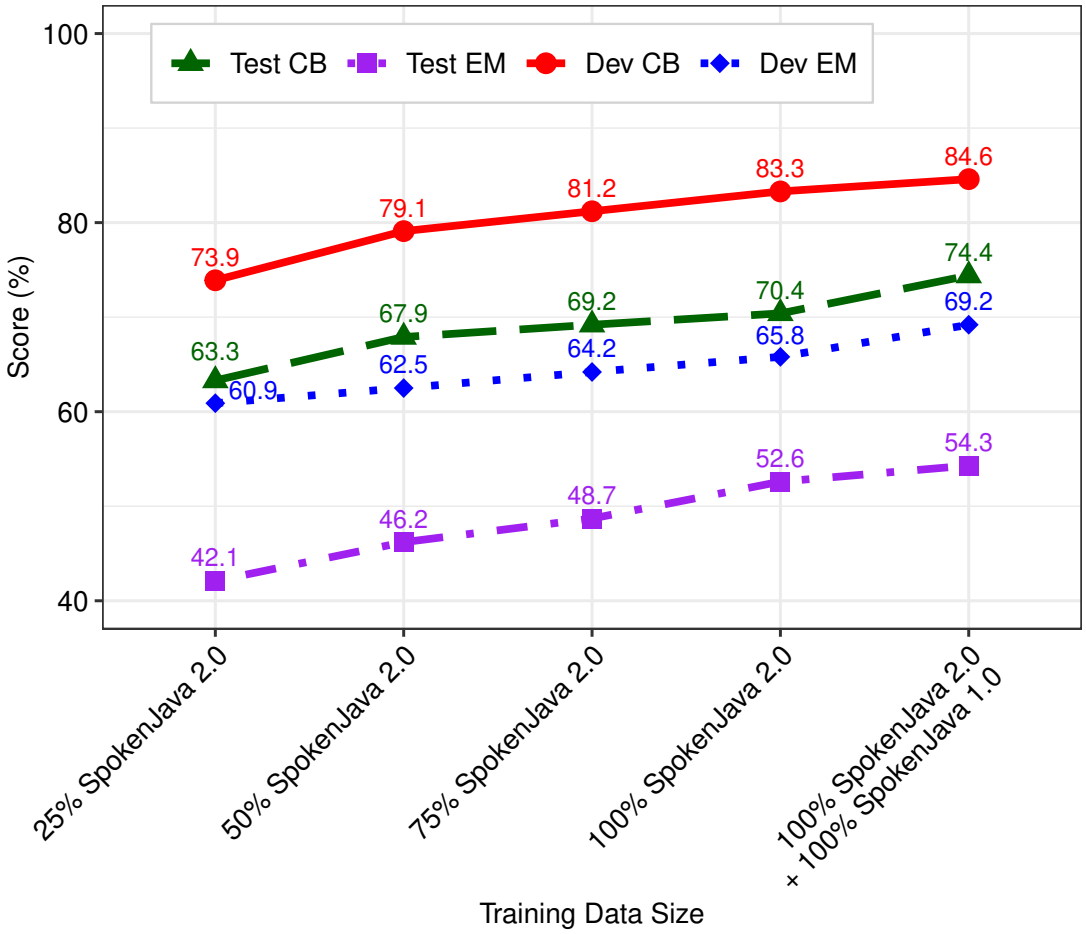


Figure 5.2: CodeBLEU (CB) scores and Exact Match (EM) accuracies of Code Llama-7b-adapted with two lines of pre- and post-code context using varied training data amounts. SpokenJava 1.0 represents data from Studies 1 and 2, while SpokenJava 2.0 represents data from Study 3 (Chapter 3).

5.5 Discussion and Limitations

In this chapter, we aimed to generate single-line Java programs from single-line spoken Java transcripts, which poses unique challenges compared to generating

code from written comments. Spoken language often lacks the structure and clarity in written text, making the conversion process more complex. Despite these challenges, our experiments demonstrated that fine-tuning a large language model with surrounding context significantly improved model performance. To our knowledge, this is the first work to adapt a model specifically for generating single-line Java programs from spoken Java transcripts.

Our best-performing model, which utilized both preceding and following context, achieved the highest CodeBLEU and exact match accuracy, demonstrating the importance of context in generating accurate code. For instance, the model's performance on correctly understanding variable names improved from 29.1% to 85.4% when incorporating context. Although our results indicate that using both preceding and following context yields the best performance, all three models — no context, pre-context, pre- and post-context — outperformed the base model. Depending on the coding scenario, each model can be utilized in a real system. For instance, the no-context model can be used when a programmer starts writing new code; the preceding-context model can be helpful as the programmer continues writing and requires an understanding of the preceding lines, and the both-context model is ideal for editing code in the middle of a block where both preceding and following context are available.

Despite the encouraging results, our best model occasionally misunderstood method names as variable names when the spoken transcript did not indicate whether it was a method or a variable. For example, given the target: `"final boolean isIdle()"` and the transcript: `"final boolean is idle"`, the model pro-

duced `"final boolean isIdle;"`. This indicates the model still struggles with certain ambiguities inherent in spoken code. This might be due to the lack of useful context that could help accurately translate such instances.

Our results showed marked success in translating recognized transcripts from a speech recognizer to the target code. The best model achieved an exact match translation 28.8% of the time, even when the transcripts contained misrecognition errors.

We focused solely on Java in this research. The fundamental challenges of translating text to code are common across programming languages. The techniques and insights gained from this study have the potential to be adapted and extended to other programming languages. Future research should explore similar studies with other programming languages to validate and refine the approach, ensuring the generalizability of our findings and contributing to the broader field of voice programming.

The dataset used for training consisted of 833 lines, whereas large code generation models are typically trained with billions or trillions of training examples. This limited size may not fully capture the variability and complexity of spoken programs. Future work should focus on expanding the dataset by collecting more spoken programs from a diverse set of speakers and with different constructs. This would help the model generalize better to different speaking styles and accents.

Data augmentation techniques can be employed to address the limitations of the small dataset. Paraphrasing spoken transcripts to simulate different ways people phrase the same instruction can enrich the dataset. Additionally, generating

synthetic data through back-translation, where large language models convert the target code back to spoken language, can mimic the structure and variability of human speech. These experiments require extensive computational power due to several factors. For back-translation, each instance requires two passes through the model—one to translate the target code back to a reference text and another to translate it back into code. This process can be particularly demanding when dealing with large datasets.

Despite using 4-bit quantization to reduce memory usage, we were limited to a batch size of 4 for the training model with surrounding contexts. This constraint significantly restricted our ability to experiment with larger batches and more complex models, which often impact the performance of a model.

5.6 Conclusion

In conclusion, our research demonstrates the feasibility and potential of generating single-line Java code from spoken language, completing the final step of our two-step pipeline. To our knowledge, this work is the first to adapt a model to generate single-line Java code from spoken language instead of written comments. By fine-tuning a large language model, with both preceding and following context, we achieved a remarkable 46.4% increase in CodeBLEU relative to the base model originally trained to generate code snippets from comments.

Despite some limitations, such as the small dataset size and computational constraints, our results are promising and lay a solid foundation for future advance-

ments in translating spoken transcripts into target lines of code. Our approach of single-line code generation might be beneficial in real-world programming scenarios where programmers input code line-by-line. It also reduces the likelihood of errors that can occur when generating larger blocks of code. The ability to generate code from spoken language can significantly enhance the accessibility and ease of programming, particularly for those who may find traditional coding interfaces challenging.

Chapter 6

Conclusion

6.1 Discussion

This dissertation presents a methodology for creating a voice programming system aimed at making programming more accessible for individuals with motor impairments and reducing the risk of Repetitive Strain Injury (RSI) for all programmers. The research journey is detailed through various stages, each contributing to improving voice programming systems using a two-step pipeline.

In Chapter 2, we provided a comprehensive understanding of the challenges faced by motor-impaired programmers. Through interviews, we gained valuable insights into their needs and preferences, which can guide the development of more accessible and effective voice programming systems. Participants expressed frustration with current systems that require learning many commands and showed a desire for a naturally spoken programming system. Participants highlighted the

need and importance of a voice programming system with one participant stating, "A voice programming system would be useful for people who are on the verge of developing RSI and also for people who can't type in the first place."

With insights from Chapter 2, Chapter 3 investigated how diverse programmers naturally speak code without learning specific commands. By analyzing the vocalization of code by novice and expert programmers, we uncovered new insights into how programmers speak code without adhering to rules. We found that participants spoke both literally and naturally. One participant noted, "Speaking without strict rules feels more natural, but if common commands (loops and method declarations) had set spoken syntax, there would be less confusion on what is going to be declared." Our user studies led to creating a dataset that includes single lines of spoken Java programs, corresponding target lines of code, and transcripts of how programmers vocalized those lines. To our knowledge, this is the first dataset of this nature, as traditional datasets used in programming language processing typically include written code or annotated text-based comments. The single-line spoken dataset captures the nuances of how programmers speak code and can be used for speech recognition, text-to-code, and code-to-text translation tasks.

In Chapter 4, we addressed the first step of our two-step pipeline: recognizing the literal words spoken by programmers. We demonstrated how to improve the recognition accuracy of spoken programs by adapting large pre-trained models. Our experiments showed that fine-tuning the model with domain-specific spoken program data and decoding with a transformer model trained with domain-specific

spoken Java transcripts resulted in substantial improvements in recognition accuracy. Our best-adapted model achieved an error rate of 4.5%, while the base model had a high error rate of 22.3% on our test set. This chapter established the effectiveness of adapted speech recognition models for accurately recognizing spoken programming languages.

Chapter 5 presents the second step of the pipeline: converting recognized transcripts of spoken Java code into the actual line of code. By adapting a large language model to understand and utilize context, we significantly improved the CodeBLEU score of the generated single-line code. We considered three scenarios: a no-context model, a pre-context model with two lines, and a both-context model surrounding two lines. The no-context model is particularly useful when a programmer writes new code without existing lines. The preceding-context model leverages the context of preceding lines of code, making it ideal for extending or continuing existing code. The both-context model, trained with preceding and following contexts, is well-suited for editing or inserting code within an existing block. Each of these models demonstrated superior performance compared to the base model. The no-context, pre-context, and pre-and post-context models showed a relative increase of approximately 33.4%, 43.2%, and 46.4% in CodeBLEU scores, respectively. Moreover, our best model with surrounding context performed well even when the recognition transcripts contained errors from the speech recognizer, achieving a relative increase of 42.8% in CodeBLEU scores compared to the base model. However, when translating misrecognized text, the CodeBLEU score was slightly lower by 8% relative to translating human transcripts.

Our study demonstrated the effectiveness of a two-step pipeline approach, where speech recognition and text-to-code translation are handled separately. This modular design allows for the independent optimization of each component. For instance, a more accurate speech recognition model can be integrated without altering the text-to-code translation stage. This flexibility enables continual improvements and the incorporation of the latest advancements in each field.

The two-step pipeline developed in this research offers flexibility in swapping different models for each stage, which can be particularly beneficial as new and improved models become available. The first step, recognizing spoken words, and the second step, converting recognized text into code, can each be optimized independently. This modular approach allows for continual improvements in each component without requiring a complete system overhaul. For instance, if a more accurate speech recognition model is developed, it can be integrated into the pipeline without altering the text-to-code conversion stage, and vice versa. The text-to-code translation model could also be adapted to generate code in different programming languages.

6.2 Future Work and Limitations

One limitation of our work was the small dataset size. Our SpokenJava 1.0 dataset consisted of 269/27/28 utterances, 29/6/6 users, and 60/5.7/6.9 minutes of audio for the train/dev/test sets, respectively. The SpokenJava 2.0 dataset consisted of 833/115/295 utterances, 50/7/10 users, and 200/25/30 minutes of audio for the

train/dev/test sets, respectively. This limited data restricted the variability and complexity that the model could learn from. In contrast, large language models like GPT-3 [61], Gemini [55], and LlaMA [62] are trained on hundreds of gigabytes of text data, encompassing billions of tokens. Employing data augmentation techniques, such as generating synthetic data, could help mitigate the limitations of a small dataset. Another promising approach for expanding the dataset involves scaling up through real-world usage. As the system is used in various environments, it can automatically gather diverse spoken Java transcripts from actual users.

We had a very small number of programmers involved in our studies. Recruiting a larger, more diverse group of participants proved challenging because all participants needed to know Java and the labor of transcription. This requirement limited our pool of potential participants. Future work could leverage platforms such as Prolific, Amazon Mechanical Turk (MTurk), and other crowdsourcing platforms to recruit more participants. Expanding the participant pool would provide more diverse data and improve the robustness of the voice programming system. Transcribing spoken data manually is a time-consuming and labor-intensive process. Future work could explore semi-automated or fully automated transcription methods to address this. Leveraging existing speech-to-text systems to produce initial transcripts, followed by human verification and correction, could reduce the transcription workload. Recruiting more transcribers through platforms like MTurk could also help scale the transcription process efficiently.

While the two-step pipeline has its advantages, an end-to-end model could simplify the system by handling the entire process in one unified step. The model can learn directly from spoken words to code without needing an intermediate step, which might result in better overall performance. The direct flow of information can allow the model to optimize the entire process as a whole, potentially capturing subtle relationships that a two-step approach might miss. However, such a model would require a much larger and more diverse dataset to train effectively, as it would need to learn both tasks at once. Additionally, end-to-end models can be more difficult to debug and optimize since errors in one part of the process can affect the entire output.

We aim to develop a fully working system in the future based on the methodology and model we developed in this work. Such a system can learn and adapt to individual programmer preferences, including variable name choices. Based on our data, we observed that people sometimes speak code literally and sometimes naturally. A dynamic model that can learn and switch between these modes based on context or user history could be highly effective. For complex statements, we found users' preference to speak literally. For instance, integrating multiple models that specialize in different aspects—such as one optimized for literal speech and another for natural speech could based on the detected speech pattern or user preference, improving the overall accuracy and user experience. It could learn the user's coding style, preferred variable names, and commonly used code patterns, providing a personalized coding experience.

6.3 Final Remarks

Our contributions lay the groundwork for creating a voice-based programming system by demonstrating the feasibility of a two-step pipeline and emphasizing the need for a flexible, adaptable approach that can evolve with user needs. We gained valuable insights into the challenges of motor-impaired programmers, explored how programmers naturally speak code, and created a novel dataset capturing diverse speech patterns. We also improved speech recognition accuracy by adapting a large pre-trained language model in the first step. In the second step, we demonstrated the effectiveness of the context in enhancing text-to-code translation. Building on our findings, future work should focus on expanding the dataset and integrating context-aware models that adapt to user preferences. The insights and methodologies from our work can be adapted to support other programming languages, making the approach versatile and widely applicable. This adaptable framework could pave the way for a more inclusive and efficient spoken programming system, benefiting diverse users.

Appendix **A**

A.1 Interview Study Questionnaire

7 motor-impaired programmers took part in your interview study presented in Chapter 2. Figures A.1 and A.2 display the pre-interview questionnaire and the open-ended questions we asked during the interviews, respectively.

Age _____ **Gender** _____

What is your current job title: _____

How much do you agree or disagree with the following statements (**X a single circle**)?

	Strongly disagree 1 2 3 4 5 6 7 Strongly agree
I consider myself a native speaker of **English**	o o o o o o o
When I speak English, **people** sometimes have trouble **understanding** me	o o o o o o o
I frequently use **speech interfaces** to control a computer	o o o o o o o
When I speak English, **computer interfaces** (e.g. Siri, Alexa, Google Assistant) sometimes have trouble **understanding** me	o o o o o o o
I consider myself as an **expert programmer**	o o o o o o o
I frequently **write programs**	o o o o o o o

Figure A.1: Pre-interview questionnaire for the interview study presented in Chapter 2.

1. **Experience with Voice User Interfaces**

- Please describe any voice user interfaces (e.g., Siri, Alexa) you use.
- How often do you use them?
- What do you use them for?
- Do you have any difficulties using them?
- What do you like/dislike about them?

2. **Programming Experience**

- Approximately how many years have you been programming?
- What languages do you program in? Rate your proficiency in each (1 = novice, 5 = expert).
- What software development tools do you usually use (e.g., IDEs)?
- What aspects of software development do you find most challenging?
- How often do you program and for how long at a time?

3. **Physical Input with Computer**

- Do you have any conditions that make it difficult for you to use a typical desktop computer? If so, are these conditions temporary or permanent?
- What input devices or methods do you use to program (e.g., mouse, keyboard, speech)?
- Have you ever tried to program by voice using speech recognition software?
    - If yes, follow-up questions:
        - What software did you use?
        - How did it work?
        - What worked well?
        - What didn't work well?
        - What do you think could be improved?
    - If no, follow-up questions:
        - What prevented you from trying programming via speech recognition?

4. **Imagining How to Do Voice Coding**

- Imagine you are using an intelligent, highly accurate, future programming by voice system. Describe how you might develop a small program using this system.
- What parts of programs do you think would be easy to input via voice?
- What parts of programs do you think would be difficult to input via voice?
- How do you envision debugging and editing an existing program via voice?
- Do you have any privacy or social concerns regarding using a voice interface to write code?
- For whom, or in what circumstances, do you envision programming by voice being most useful?

Figure A.2: Open-ended questions for the interview study presented in Chapter 2.

A.2 Study 1 and Study 2 Questionnaire

Figures A.3 and A.4 show the pre-experiment and post-experiment questionnaires used in Study 1 and Study 2 presented in Section 3.2 and Section 3.3.

Age ☐ Gender ☐ Programming experience e.g. 2 **years.**

How much do you agree or disagree with the following statements (**X a single circle**)

#	Statements	Strongly disagree 1 2 3 4 5 6 7 Strongly agree
1	I consider myself a **fluent speaker of English**	o o o o o o o
2	When I speak English, I have a **foreign accent**	o o o o o o o
3	When I speak English, people sometimes have **trouble understanding me**	o o o o o o o
4	I frequently **use speech interfaces** to control a computer	o o o o o o o
5	When I speak English, **voice assistants** (e.g., Siri, Alexa, Google Assistant) sometimes have **trouble understanding me**	o o o o o o o
6	I consider myself an **expert programmer**	o o o o o o o
7	I frequently **write programs**	o o o o o o o
8	I frequently use **voice to input programs**	o o o o o o o

9. Do you have any conditions that make it difficult for you to use a typical desktop computer? If so, what are these conditions?

Figure A.3: Pre-experiment questionnaire used in Study 1 and Study 2 presented in Section 3.2 and Section 3.3.

How much do you agree or disagree with the following statements (**X a single circle**)?

#	Statements	Strongly disagree 1 2 3 4 5 6 7 Strongly agree
1	I found speaking variables with **one word easy (e.g., result)**	○ ○ ○ ○ ○ ○ ○
2	I found speaking variables with **multiple words easy (e.g., firstname)**	○ ○ ○ ○ ○ ○ ○
3	I found speaking variables with **abbreviated words (e.g., lrgNumCntr)**	○ ○ ○ ○ ○ ○ ○
3	I found speaking variables with **mixed case easy (e.g., lastCarModel)**	○ ○ ○ ○ ○ ○ ○
4	I found speaking **a few lines** in a program easy	○ ○ ○ ○ ○ ○ ○
5	I found speaking **the whole program easy**	○ ○ ○ ○ ○ ○ ○
6	I found the given programs **easy to understand**	○ ○ ○ ○ ○ ○ ○

7. Give up a few **examples of variables** you were most **uncertain** about how to speak.
8. What **parts of code** were you most **uncertain** about how to speak?
9. Do you have any **suggestions/comments** about the experiment?

Figure A.4: Post-experiment questionnaire used in Study 1 and Study 2 presented in Section 3.2 and Section 3.3.

A.3 Study 3 Questionnaire

Figures A.5 and A.6 present questionnaires used in Study 3 presented in Section 3.4.

Age ☐ Gender ☐ Programming experience e.g. 2 **years.**

How much do you agree or disagree with the following statements (**X a single circle**)

#	Statements	Strongly disagree 1 2 3 4 5 6 7 Strongly agree
1	I consider myself a **fluent speaker of English**	o o o o o o o
2	When I speak English, I have a **foreign accent**	o o o o o o o
3	When I speak English, people sometimes have **trouble understanding me**	o o o o o o o
4	I frequently **use speech interfaces** to control a computer	o o o o o o o
5	When I speak English, **voice assistants** (e.g., Siri, Alexa, Google Assistant) sometimes have **trouble understanding me**	o o o o o o o
6	I consider myself an **expert programmer**	o o o o o o o
7	I frequently **write programs**	o o o o o o o
8	I frequently use **voice to input programs**	o o o o o o o

9. What type of microphone are you using for this study (e.g. wired headset, build-in laptop microphone, wireless earpods, etc.)?

10. Do you have any conditions that make it difficult for you to use a typical desktop computer? If so, what are these conditions?

Figure A.5: Pre-experiment questionnaire used in Study 3 presented in Section 3.4.

How much do you agree or disagree with the following statements (**X a single circle**)?

#	Statements	Strongly disagree 1 2 3 4 5 6 7 Strongly agree
1	I found speaking a **variable name** easy.	○ ○ ○ ○ ○ ○ ○
2	I found speaking a **conditional statement (if, else)** easy.	○ ○ ○ ○ ○ ○ ○
3	I found speaking a **loop statement (for, while)** easy.	○ ○ ○ ○ ○ ○ ○
4	I found speaking a **method signature/method call** easy.	○ ○ ○ ○ ○ ○ ○
5	I found speaking **comments** easy.	○ ○ ○ ○ ○ ○ ○
6	I found speaking a **math statement** easy.	○ ○ ○ ○ ○ ○ ○

7. What **types of code** (e.g. variables, loop, method) were you most uncertain about what to speak and why?

8. Did you find it preferable to speak with no strict rules or would you rather learn specific rules for spoken commands?

9. Imagine you are writing a fairly complex line of code (e.g. an if-statement with multiple conditions that involve calling functions). How would you envision a speech interface working in order to support entering such a line?

10. Do you have any **suggestions/comments** about the experiment?

Figure A.6: Post-experiment questionnaire used in Study 3 presented in Section 3.4.

Appendix B

B.1 Google Speech-to-Text Experiment Details

Table B.1 shows the Word Error Rate (WER) for different n-gram language models we investigated in the Google speech-to-text language model adaptation experiment presented in Section 4.3.

Language model	WER
2-gram	20.8
3-gram	22.8
4-gram	23.5
5-gram	23.9

Table B.1: Word Error Rate (WER) for different n-gram language models in the experiment with Google's speech-to-text presented in Section 4.3.

B.2 N-gram Language Modeling Results

Table B.2 shows the perplexity for all trigram to 8-gram models trained on different datasets in our language modeling experiment presented in Section 4.4.3. The best language model was a 4-gram mixture model with a perplexity of 22.1.

N-gram order	Training dataset		
	SpokenJava	CodexGlue	LibriSpeech
3	16.6	2420.6	17215.0
4	15.4	2310.7	14983.0
5	16.3	2311.9	14983.2
6	17.6	2323.7	14999.5
7	18.1	2322.0	15527.8
8	18.5	2339.2	15591.2

Table B.2: Perplexities for different n-gram language models trained on SpokenJava 1.0, CodexGlue single line comments, and LibriSpeech datasets presented in Section 4.4.3.

B.3 Adapted wav2vec Model Hyperparameters

Table B.3 shows the beam search decoding parameters for the wav2vec models presented in Section 4.4.4.

Model	LM	LM weight	WIP
LibriSpeech (960h)	4-gram-libri	0.3143	-0.6010
LibriSpeech (960h)	4-gram-mixture	2.3922	2.6103
LibriSpeech (960h) + SpokenJava (1h)	4-gram-libri	0.9013	1.8621
LibriSpeech (960h) + SpokenJava (1h)	4-gram-mixture	1.1311	0.2042

Table B.3: Beam search decoding parameters for all fine-tuned models presented in Section 4.4.4. LM = Language Model, WIP = Word Insertion Penalty.

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2021.naacl-main.211.

[2] Rosana Ardila, Megan Branson, Kelly Davis, Michael Kohler, Josh Meyer, Michael Henretty, Reuben Morais, Lindsay Saunders, Francis Tyers, and Gregor Weber. Common voice: A massively-multilingual speech corpus. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 4218–4222, Marseille, France, May 2020. European Language Resources Association. ISBN 979-10-95546-34-4. URL https://aclanthology.org/2020.lrec-1.520.

[3] Stephen C Arnold, Leo Mark, and John Goldthwaite. Programming by voice, vocalprogramming. In *Proceedings of the fourth international ACM conference on Assistive technologies*, pages 149–155. ACM, 2000.

[4] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems*, 33:12449–12460, 2020.

[5] Alexei Baevski, Wei-Ning Hsu, Alexis Conneau, and Michael Auli. Unsupervised speech recognition. *Advances in Neural Information Processing Systems*, 34:27826–27839, 2021.

[6] Andrew Begel and Susan L Graham. Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 99–106. IEEE, 2005.

[7] Andrew Begel and Susan L Graham. An assessment of a speech-based programming environment. In *Visual LAnguages and Human-Centric Computing (VL/HCC'06)*, pages 116–120. IEEE, 2006.

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman et al. Evaluating large language models trained on code. 2021.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[13] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018. IEEE, 2019.

[14] Yu-An Chung and James Glass. Generative pre-training for speech with autoregressive predictive coding. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3497–3501. IEEE, 2020.

[15] Alain Désilets. Voicegrip: A tool for programming-by-voice. *International Journal of Speech Technology*, 4(2):103–116, 2001.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[17] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of*

the Association for Computational Linguistics: EMNLP 2020, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[20] Marios Flourentzou, Eirini C. Schiza, and Constantinos Pattichis. A case study of voice recognition technology for developing programming skills for students with motor disabilities. PETRA '24, page 682–683, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717604. doi: 10.1145/3652037.3663901. URL https://doi.org/10.1145/3652037.3663901.

[21] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv:2101.00027*, 2020.

[22] Benjamin M Gordon. Developing a language for spoken programming. In *Sixteenth AAAI/SIGART Doctoral Consortium*, 2011.

[23] Cassandra Guy, Michael Jurka, Steven Stanek, and Richard Fateman. Math speak & write, a computer program to read and hear mathematical input. *University of California Berkeley*, 2004.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] Hongzhao Huang and Fuchun Peng. An empirical study of efficient asr rescoring with transformers, 2019. URL https://arxiv.org/abs/1910.11450.

[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. URL https://arxiv.org/abs/1909.09436.

[27] Hyunhoon Jung, Hee Jae Kim, Seongeun So, Jinjoong Kim, and Changhoon Oh. Turtletalk: an educational programming game for children with voice user interface. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–6, 2019.

[28] Jacob Kahn, Ann Lee, and Awni Hannun. Self-training for end-to-end speech recognition. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7084–7088. IEEE, 2020.

[29] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2, 2019.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[31] Bo Li, Tara N. Sainath, Ruoming Pang, and Zelin Wu. Semi-supervised training for end-to-end models via weak distillation. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2837–2841, 2019. doi: 10.1109/ICASSP.2019.8682172.

[32] Ke Li, Zhe Liu, Tianxing He, Hongzhao Huang, Fuchun Peng, Daniel Povey, and Sanjeev Khudanpur. An empirical study of transformer-based neural language model adaptation. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7934–7938, 2020. doi: 10.1109/ICASSP40776.2020.9053399.

[33] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, 2016.

[34] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. URL https://arxiv.org/abs/2102.04664.

[35] Rinor S. Maloku and Besart Xh. Pllana. Hypercode: Voice aided programming. *IFAC-PapersOnLine*, 49(29):263–268, 2016. ISSN 2405-

8963. doi: https://doi.org/10.1016/j.ifacol.2016.11.073. URL https://www.sciencedirect.com/science/article/pii/S2405896316325095.

[36] Paul W McBurney and Collin McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2015.

[37] Jonathan Giovanni Soto Muñoz, Arturo Iván de Casso Verdugo, Eliseo Geraldo González, Jesús Andrés Sandoval Bringas, and Miguel Parra Garcia. Programming by voice assistance tool for physical impairment patients classified in to peripheral neuropathy centered on arms or hands movement difficulty. In *2019 International Conference on Inclusive Technologies and Education (CONTIE)*, pages 210–2107. IEEE, 2019.

[38] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023. URL https://arxiv.org/abs/2203.13474.

[39] Anna Nowogrodzki. Speaking in code: how to program by voice. *Nature*, 559(2):141–142, 2018.

[40] Sadia Nowrin. Programming by voice. *ACM SIGACCESS Accessibility and Computing*, (132):1–1, 2022.

[41] Sadia Nowrin and Keith Vertanen. Programming by Voice: Exploring User Preferences and Speaking Styles. In *Proceedings of the 5th International*

Conference on Conversational User Interfaces, CUI '23, pages 1–13, New York, NY, USA, July 2023. Association for Computing Machinery. ISBN 9798400700149. doi: 10.1145/3571884.3597130. URL https://dl.acm.org/doi/10.1145/3571884.3597130.

[42] Sadia Nowrin and Keith Vertanen. Leveraging large pretrained models for line-by-line spoken program recognition. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 12216–12220. IEEE, 2024.

[43] Sadia Nowrin, Patricia Ordóñez, and Keith Vertanen. Exploring motor-impaired programmers' use of speech recognition. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392587. doi: 10.1145/3517428.3550392. URL https://doi.org/10.1145/3517428.3550392.

[44] Obianuju Okafor and Stephanie Ludi. Voice-enabled blockly: Usability impressions of a speech-driven block-based programming system. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '22, pages 1–5, 2022.

[45] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*

(Demonstrations), pages 48–53, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-4009. URL https://aclanthology.org/N19-4009.

[46] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015. doi: 10.1109/ICASSP.2015.7178964.

[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[48] Vineel Pratap, Awni Hannun, Qiantong Xu, Jeff Cai, Jacob Kahn, Gabriel Synnaeve, Vitaliy Liptchinsky, and Ronan Collobert. Wav2letter++: A fast open-source speech recognition system. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6460–6464. IEEE, 2019.

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[50] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a

method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

[51] Lucas Rosenblatt, Patrick Carrington, Kotaro Hara, and Jeffrey P Bigham. Vocal programming for people with upper-body motor impairments. In *Proceedings of the Internet of Accessible Things*, page 30. ACM, 2018.

[52] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611, 2020.

[53] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv:2308.12950*, 2023.

[54] Tavis Rudd. Using python to code by voice. http://pyvideo.org/video/1735/using-python-to-code-by-voice, 2013.

[55] Hamid Reza Saeidnia. Welcome to the gemini era: Google deepmind and the information industry. *Library Hi Tech News*, (ahead-of-print), 2023.

[56] Waseem Sheikh, Dave Schleppenbach, and Dennis Leas. Mathspeak: a non-ambiguous language for audio rendering of mathml. *International Journal of Learning Technology*, 13(1):3–25, 2018.

[57] Yuanfeng Song, Raymond Chi-Wing Wong, and Xuefang Zhao. Speech-to-sql: toward speech-driven sql query generation from natural language question. *The VLDB Journal*, pages 1–23, 2024.

[58] Dragon Naturally Speaking. Dragon naturally speaking. *Retrieved June*, 21:2022, 2022.

[59] Andreas Stolcke. SRILM - An extensible language modeling toolkit. In *Proc. 7th International Conference on Spoken Language Processing (ICSLP 2002)*, pages 901–904, 2002. doi: 10.21437/ICSLP.2002-303.

[60] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1433–1443, 2020.

[61] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023. URL https://arxiv.org/abs/2304.11938.

[62] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, and Yasmine Babaei et al. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

[63] Jessica Van Brummelen, Kevin Weng, Phoebe Lin, and Catherine Yeo. CONVO: What does conversational programming need? In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5. IEEE, 2020.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[65] Amber Wagner and Jeff Gray. An empirical evaluation of a vocal user interface for programming by voice. *International Journal of Information Technologies and Systems Approach (IJITSA)*, 8(2):47–63, 2015.

[66] Amber Wagner, Ramaraju Rudraraju, Srinivasa Datla, Avishek Banerjee, Mandar Sudame, and Jeff Gray. Programming by voice: A hands-free approach for motorically challenged children. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, page 2087–2092, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310161. doi: 10.1145/2212776.2223757. URL https://doi.org/10.1145/2212776.2223757.

[67] Angela M Wigmore, Gordon JA Hunter, Eckhard Pflügel, and James Denholm-Price. Talkmaths: A speech user interface for dictating mathematical expressions into electronic documents. In *International workshop on speech and language technology in education*, 2009.

[68] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's

transformers: State-of-the-art natural language processing, 2020. URL https://arxiv.org/abs/1910.03771.

[69] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, 2017.

[70] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022. URL https://arxiv.org/abs/2205.01068.

[71] Guolin Zheng, Yubei Xiao, Ke Gong, Pan Zhou, Xiaodan Liang, and Liang Lin. Wav-BERT: Cooperative acoustic and linguistic representation learning for low-resource speech recognition. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2765–2777, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.236. URL https://aclanthology.org/2021.findings-emnlp.236.