7-1-2013

Improving Spoken Programming Through Language Design and the Incorporation of Dynamic Context

Benjamin M. Gordon

Benjamin M. Gordon

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

George Luger , Chairperson

Darko Stefanovic

Caroline Smith

Shuang Luan

Improving Spoken Programming Through Language Design and the Incorporation of Dynamic Context

by

Benjamin M. Gordon

B.S., New Mexico Institute of Mining and Technology, 2003
M.S., Computer Science, University of New Mexico, 2010

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2013

Dedication

To my beloved Bailee. Without your endless patience and encouragement, this never would have been finished.

Acknowledgments

I would like to thank my advisor, Professor George Luger, for his support and encouragement. Thank you for all the times you repeated the advice I needed as many times as necessary for me to catch on to what you were saying.

I would also like to thank Leader Technologies for their financial support. This took much longer than anybody there ever realized they were signing up for, but they never wavered in their decision to see it through to the end.

Improving Spoken Programming Through Language Design and the Incorporation of Dynamic Context

by

Benjamin M. Gordon

B.S., New Mexico Institute of Mining and Technology, 2003
M.S., Computer Science, University of New Mexico, 2010

PhD, Computer Science, University of New Mexico, 2013

Abstract

Spoken programming refers to the idea that human speech can be used as input (via dictation) to create the text of a program. Most previous research on spoken programming has focused on enabling diction of text in existing programming languages, either by creating a spoken syntax for the existing language or by attempting to extract the meaning from general English and using it to produce a program. We take an alternate approach of modifying the programming language syntax to support speech: We have created a new programming language that uses English words and phrases for its syntax. This provides several benefits to the spoken programming process, including easier reuse of existing English-based speech tools and a more direct mapping from user speech onto the visible program text.

Programming is not just about the syntax of the language; the programming environment is also an important part of the process. In English prose speech recognition, knowledge of the context of the conversation or document being produced is useful to disambiguate otherwise error-prone phrases or words. Our spoken programming environment explores the same idea applied to programming by incorporating several forms of context inherent in computer programs to guide and enhance the speech recognition engine: We use the structure and grammar of the language itself, we use scoping of symbols and identifiers, and we incorporate constraints gleaned from type inference. Together, these sources of context allow our programming environment to further improve on the initial performance gain enabled by our new syntax.

In this dissertation, we describe the theory and design decisions that went into the aforementioned programming language and its supporting programming environment, as well as the implementation details. We discuss how this setup fits in with—and differs from—previous research in spoken programming. To test the effectiveness of this environment, we conducted a small user study, which is described. The results of this study are presented, together with some qualitative exploration of their implications. Finally, we conclude with some speculation about how the ideas presented here could fit into programming as a whole and suggestions for future research in this direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ever since the invention of general-purpose computers, entering programs into the system has been a vital task. Over time, we advanced from setting banks of switches to loading stacks of punch cards (which indirectly represent keyboard input, since the punch card creating machines used keyboards), finally settling on direct keyboard input. Modern programming environments provide GUIs and a few mouse-based features, but the predominant mechanism for entering the program text remains the keyboard. This keyboard-based paradigm has served programmers well for decades now, and this research does not propose to do away with it. However, it is not ideal in all situations.

1.1 Inadequacy of Keyboards

Keyboards seem to have been doing fine for many years as an input device. Many programmers can type upwards of 60 words per minute, and the speed of input isn't really the bottleneck for most people's ability to solve problems with a program. So why would we want to consider alternatives? First, it is important to expand accessibility to existing computing devices. Undoubtedly, everyone can think of a

time when they had to balance a keyboard on an awkward surface or make a mouse work on a non-flat surface.

Furthermore, even when an ideal setup is available, people may have many reasons to prefer to avoid their keyboard. As a concrete example, many people who have repetitive stress injuries (RSI), carpal tunnel syndrome, or other motor impairments may experience considerable difficulty using a keyboard and mouse. Such an injury causes no impairment to a person's intelligence or cognitive abilities, leaving them in the frustrating situation of being perfectly able to think and plan programs, but unable to actually type them in. A well-known example of this situation is when Richard Stallman developed severe hand pain in the early 1990s, forcing him to dictate his work to human typists [44].

A second—perhaps more common—example of a place where the keyboard-based paradigm breaks down is tablet computing. While keyboard-less tablet computers have existed for many years, the recent introduction of the iPad and similar Android-based tablets has brought new attention to the idea. Many casual users have found that a tablet is sufficient for all of their common computing needs, and advanced users often find it to be usable for many day-to-day tasks. Today's generation of tablets are suitable for viewing many forms of content, but are much more difficult to use for creating new content.

One area where tablets are widely acknowledged to be deficient is the task of entering and manipulating large amounts of textual data. The virtual keyboards available on tablets are functional in terms of entering small amounts of text, but they leave much to be desired for use as a keyboard replacement. Newly announced products already claim support for high-resolution screens, multicore processors, and large memory capacities, but they still will not include a keyboard. It is certainly possible to pair a tablet with an external keyboard if a large amount of text entry is needed, but carrying around a separate keyboard seems to defeat the main appeal of a tablet computer.

The text entry issue is likely to be even more pronounced when entering large amounts of punctuation and non-English text, such as the text of a traditional programming language. This occurs for two reasons: the keyboard layouts on mobile devices typically include fewer non-letter keys than a standard keyboard due to the lack of space; and because the completion and correction routines built into such systems have been trained for English. Even if the problems of haptic feedback and English auto-correction are resolved, the virtual keyboard takes up a large fraction of the already limited display area, which makes it impossible to display large quantities of information at the same time as text is being entered. As tablets become more ubiquitous for everyday computing, it is only a matter of time before people begin to use them for more advanced tasks, including programming. Before this can become truly viable, we need a programming environment that supports non-keyboard programming. The research presented here does not get us all the way to a full keyboard-less programming environment, but we hope to convince the reader by the end of this dissertation that it takes an important step in the right direction.

1.2 Programming by Voice

There are many alternatives to keyboards that can be considered. Typical tablets today make substantial use of finger-based input, using multitouch and gestures to increase their capabilities. Another popular choice for touch input historically has been stylus- or pen-based input mechanisms. With the recent rise of embedded cameras and peripherals like Microsoft Kinect [28] (or even the much older "put-that-there" project from MIT [7]), one might even incorporating some sort of physical gestures into programming. These ideas are all interesting research areas in their own rights. This dissertation, however, will focus on speech. In particular, we will show that speech can be used as a viable primary input mechanism for programming. We will come back to these other ideas briefly in Chapter 7 when we speculate on

how they might be combined with speech.

Once one has decided to program via speech input, one must choose a language to program in. The previous research described in Chapter 2 has largely accepted the syntax of existing computer languages as a given, choosing to focus on turning human speech into the existing languages. We claim that better results may be achievable by taking human speech as the primary consideration and modifying the computer language instead. To this end, a primary consideration of this dissertation is the creation of a new programming language that is designed to mimic human (English) speech syntax.

Productive programming requires more than merely a convenient syntax. An additional important factor is an editor that is optimized for the assumption of voice control instead of keyboard and mouse input. Most people would find the idea of using the same type of editor to write a memo and edit a photo to be a strange one. Similarly, why should we expect that merely adding a few voice commands to a primarily keyboard-driven editor will produce an excellent voice-driven editor? At minimum, it must be possible to dictate new code as well as edit existing code with minimal use of a keyboard or mouse. This by itself would not constitute anything new. The contribution of this dissertation is the confirmation that the programming environment can incorporate additional contextual information about the programming task to improve the performance of speech recognition in the programming domain like it does in other traditional areas where English speech recognition has been used.

The first form of context we incorporate is the structure imposed by the formal grammar of the language itself. Unlike English conversational speech, where grammatical rules are largely conventions that can be stretched or broken as long as communication is not impaired, the syntax of programming languages is fully described and circumscribed by a formal grammar. Sentences that are ungrammatical are also in error. Both the form of individual statements and the ways they can

be combined to form a nested block structure are dictated by the grammar, and we make use of this structure to let the recognizer understand which sentences are implausible or impossible at any particular point in the program.

Our use of the program structure goes beyond merely requiring that dictated statements conform to a context-free grammar that describes the language. We treat programming as a dialog between the user and the environment. Just like a dialog between two humans, the two participants have a certain context built up based on their shared expectations and the sentences that have already been uttered. In our case, the set of statements the computer is expecting to hear changes throughout the conversation based on the current state of the in-progress program and user's statements. Just as transition words and phrases help a participant in a conversation modify their internal context as the subject changes, various key statements and phrases within a program trigger a change in context as well. We implement this idea through the use of multiple grammars that are switched and modified at runtime as the user is dictating to incorporate the knowledge that is built up in the process of writing the program. Full details are provided in Section 4.2.1.

The second form of context considered in this dissertation is identifier scopes. Because our language has a nested block structure similar to C, different symbols or identifiers are visible at different places in the text. By making the recognizer aware of which symbols are in scope at the point where the user is adding statements, we can bias it away from introducing invalid references to unknown symbols and toward references to the symbols that are visible. On a more sophisticated level, we can use the principle of locality to probabilistically prefer symbols that are defined in closely-nested scopes over those that are defined at farther nesting levels. See Section 4.2.2 for a description of the implementation.

Finally, we make use of type information that is dynamically discovered through type inference. Like many modern programming languages, our spoken language does not require the user to annotate all their variables with explicit type information.

Instead, the compiler recovers it automatically when the program is compiled. Additionally, our programming environment dynamically performs limited type inference as the user is dictating the program and feeds this into the speech recognizer to bias its recognition decisions. Similar to the idea of preferring legal symbol references to undefined ones, we use the type information to prefer correctly-typed symbols references to invalid ones. The type context and grammars are updated with each use of a symbol to help the recognizer continuously adjust to the expected uses of variables and functions. See Section 4.2.3 for details.

1.3 Conclusions

We have created a new programming language with spoken input as a primary design feature instead of an after-the-fact bolt-on. This language is Turing complete and offers similar expressiveness to C. In addition, we have created a compiler for this language and an Eclipse plugin that integrates the CMU Sphinx speech recognizer to form a more complete spoken programming environment. Our programming environment improves the performance of the speech recognizer not only by using syntax that more closely matches speech, but also by incorporating additional context induced by the programming task to guide the recognizer.

We have performed a user study of people's interaction with this environment. The results show promising (and statistically significant) improvements in several areas: participants' ability to complete working programs using only speech, the speed of entering programs, and a reduction in the number of errors the system produces.

In Chapter 2, we provide an overview of previous research in the area and relevant background material. In Chapter 3, we describe the language syntax and programming environment. Chapter 4 is a description of how we implemented the theory

from Chapter 3. In Chapter 5, we describe the experimental setup for our user study. Chapter 6 gives the experiment results and discussion. Finally, Chapter 7 contains our conclusions and future work.

Chapter 2

Background

We start our discussion with background material and previous research. In addition to a brief exploration of visual programming and natural language programming, we provide an in-depth review of previous work in spoken programming. We also describe the idea of cognitive distance as a means of qualitatively thinking about the potential relative merits of different programming environments. The specific contributions of this dissertation and how they fit into previous work are described in subsequent chapters.

2.1 Alternatives to Keyboards

We first consider two alternatives to keyboards as an input mechanism. In visual programming, the user creates a program by drawing and manipulating objects with a stylus or mouse. These objects typically also include text values that can be edited with the keyboard. Depending on the implementation, visual programming can be mostly text-based or mostly drawing-based. Spoken programming is concerned instead with using speech and voice recognition as input. The type of actions and/or text produced as output varies as we will describe below. Some multi-modal systems

also combine aspects of visual and spoken programming.

2.1.1 Visual Programming

Most prior approaches to spoken programming have proceeded by adding or creating a spoken syntax for a previously existing textual or visual language. Leopold and Ambler added voice and pen control to a visual programming language called Formulate [29]. Formulate was a programming environment where the user could draw various objects on a form and set object properties to equations. The equations could refer to values from other objects. Computation proceeded evaluating the equations and updating objects appropriately. The original system was based on a keyboard and mouse. Leopold and Ambler extended the system by allowing the user to select and manipulate objects using a stylus or voice input. The system accepted a limited set of words allowing object selection, equation entry, and menu selection through voice commands. At the conclusion, the authors felt that they had built a system that was completely usable without the keyboard or mouse. However, because the main goal of their research was to investigate the combination of voice and stylus input, they omitted voice control of certain features that they felt were better suited to the stylus, such as text selection and editing.

2.1.2 Spoken Programming

In the realm of pure spoken programming, Désilets, Fox and Norton created VoiceCode [10] at the National Research Center in Canada. This system combines the Emacs text editor with a commercial voice recognition product to allow speech-based input of multiple existing programming languages. VoiceCode addresses the difference between spoken syntax and code syntax by translating a standardized high-level spoken syntax into code templates in whichever language is being dictated. This also

simplifies the speech recognizer, because only the high-level language is recognized, rather than attempting to directly speak the programming language syntax. The system can be extended for new languages by adding grammars and suitable templates, and the feasibility of this was demonstrated by Masuoka in an undergraduate thesis at the University of Maryland [31]. VoiceCode supports intermixed code entry and navigation commands, using context to disambiguate. For example, the "next one" command repeats the previous command if it was a navigation command, but is interpreted as an identifier in other circumstances. Finally, to increase usability with existing code, VoiceCode supports a novel mechanism of mapping between English identifier phrases and commonly-used programmer abbreviations. For example, the authors' example video demonstrates the system turning the spoken "Context Sensitive Command Set" into the symbol "CSCmdSet" [11]. The VoiceCode software has been released as an open source project available online [12], but the project has seen little activity and no major releases since 2006.

In his PhD dissertation, Andrew Begel created an environment for speaking Java code [6]. Begel and Graham started by studying how programmers verbalized code [4]. In their study, they asked a number of expert programmers to read small programs written in Java. This study confirmed that programmers do not simply speak every symbol on the page, but rather come up with abstractions and spoken shorthands that describe the code. They found that several of the programmers did not even attempt to speak any of the code on the page; instead they used higher-level phrases like "set all the fields of array to null." Based on this study, Begel and Graham developed a spoken variant of Java called Spoken Java. Spoken Java is explicitly not meant to be a natural-language programming language, but is "about using features of natural language to simplify the verbal input form of a conventionally designed programming language" [4]. In addition, the authors compare their syntax to VoiceCode. Because Spoken Java is meant to be Java rather than a language agnostic meta-syntax, it is considerably more concise than VoiceCode,

and may be subjectively more "natural" feeling. For example, the declaration of a simple for loop in Spoken Java takes approximately half as many words as the same declaration in VoiceCode.

In addition to the Spoken Java language syntax, Begel and Graham developed a suitable plugin (SPEED) for the Eclipse development environment to enable speech input [4, 5, 6]. The completed environment was evaluated by performing a small study with human programmers. The programmers agreed that they could use the system if they developed RSI or some other reason for a keyboard to be unavailable, but none were willing to switch to Spoken Java for their daily programming.

Arnold, Mark and Goldthwaite proposed a system called VocalProgramming [2]. Their system was intended to take a context free grammar (CFG) for a programming language and automatically create a "syntax-directed editor," meaning an editor that makes use of the programming language syntax for navigation and editing rather than merely allowing text entry. For example, they suggested that a navigation command like "move down to the next statement" would move to the next statement in the program regardless of how many lines that represented. After the initial paper, the system appears to have never been implemented, nor have any follow-on papers been published. It was described as abandoned by Begel and Graham [4].

Shaik et al. created an Eclipse plugin called SpeechClipse to permit voice control of the Eclipse environment itself [40]. They permitted dictation of "well-known programming language keywords," but primarily concentrated on providing access to the menu and keyboard commands available in Eclipse.

The approaches cited above have all tried to add direct spoken syntax for an existing language. David Price et al. took a different approach with NaturalJava [35], a system for generating Java code from natural English phrases. They started by performing a "Wizard of Oz" study of novice programmers to determine how users would interact with a programming system they could speak to [37]. This study

revealed that more than 75% of the words and phrases used to describe program constructs were common to all users [38]. Using this result, they created a system that performed shallow semantic parses on the user's input to turn it into case frames representing programming concepts. These case frames were then used to infer what editing operations the user intended to perform, and the actual operations implemented using an abstract syntax tree manager to ensure that the resulting Java code remained valid. Price's prototype was implemented using keyboard-entered text rather than spoken input. In his master's thesis [36], he focused on how the Wizard of Oz study could be used to extend the system to a full spoken programming system, but it appears to have never been fully implemented. The NaturalJava system was a step away from directly speaking the Java syntax, but it was still focused on producing Java source code as its output.

A recent talk by Tavis Rudd gives an example of doing programming by voice in the opposite, "unnatural" way [39]. He demonstrates a system that uses Python to bridge Dragon NaturallySpeaking to Emacs, Python, and other tools he uses for his daily programming. Rather than attempting to create code from English, he uses made-up nonsense words and "animal grunts" to generate key presses and run commands inside his editor, reverting to English only for identifier names, strings, comments, and other similar prose content. He reports having over 2000 commands in the system to drive all aspects of his daily workflow without use of the keyboard. This includes use of the shell and remote commands, not just writing programs. During his recovery from an RSI injury, Rudd made use of this spoken system for all of his computer use for a period of approximately a year. Even today, several years later, he reports that he uses the system for 40 to 60% of his programming.

While Rudd's system provides a fascinating example of a highly productive real-world spoken programming environment, is also open to several important criticisms. First, his system is specialized for a single person. A second programmer who wished to use the same technique would have to develop his own entire set of commands

customized for his own workflow. This is not a system that can be used by multiple people without customization, and it is not even possible for one user to train another user unless the two happen to share a significant fraction of their commands and work styles.

Second, the user of the system must memorize hundreds if not thousands of made-up, non-English nonsense words. It is like learning an entire new language. This aspect of "naturalness" will be discussed further in Section 2.3. Rudd himself estimates that it would take a person two to three months of steady effort to become proficient at the system. While this is clearly less effort than required to learn a new human language, it is a significantly longer period of time than most programmers require to become productive in a new programming language.

Outside the realm of traditional programming languages, Fateman considered the task of speaking mathematical expressions [18]. He studied a system that created TEX output from a spoken form of equations. Even this "simpler" form of programming input turned out to be fraught with difficulties. He found that conventional readings of equations are completely ambiguous. For example, the well-known quadratic formula is typically spoken without any of the information necessary to unambiguously group the elements under the radical or the fraction. Nevertheless, he found that many mathematical expressions can be recognized in the expected way without demanding explicit parenthesization by the application of some common usage rules. This dilemma is also common in programming language grammars; the "dangling-else" problem in C is a well-known example [19].

Elliott and Bilmes also attempted to add a speech-based interface to a mathematics package [17]. They started with an existing computer algebra system (CAS) and wrote a separate speech-based front-end called CamMath that converted the user's speech into virtual keyboard and mouse events to control the CAS. CamMath can be used to control the CAS and work through mathematical operations, but it cannot be used to create standalone programs.

2.2 Other "Natural" Programming

In addition to spoken programming, a number of researchers have investigated programming via written natural language text. Fuchs and Schwitter created a system called Attempto [20, 21] that converted program specification text from a restricted subset of English—Attempto Controlled English—into executable Prolog programs. The system supplies a predefined set of rules for English grammar and word knowledge about common pronouns, articles, adverbs, etc. Users then supply domain-specific sets of content words that are relevant for their applications. Attempto parses the user's input in a deterministic manner and displays its interpretation of the text to the user. If the user is dissatisfied with the result, he must reword his input so that Attempto finds a different parse. Once the specification has been converted to Prolog, the system also permits the user to enter queries in ACE and converts the Prolog output back to ACE. Thus, the user appears to interact with the system entirely using ACE and never needs to be aware that Prolog is used behind the scenes. Nevertheless, the user's text is never directly compiled into an executable. Attempto has seen ongoing development and is currently at version 6.6 [3].

In the object-oriented domain, Bryant et al. have developed a system of translating natural-language program specifications into formal specifications, and from there into executable code [8]. Like Attempto, Bryant's system was also based on a subset of written natural language. Furthermore, the user does not enter the code that will be compiled. Instead, the user enters a set of declarative statements that create a specification for the program. The system then creates the skeleton of a program that implements the specification, but it does not correspond directly to anything the user has typed.

2.3 Evaluation of Programming Environments

The idea that some programming environments might be better than others has probably been around since machine code programmers were watching the first assemblers and Fortran compilers produce code. However, most ideas of productivity and effectiveness of these environments have come down to intuitive ideas and empirical studies rather than formal theories. One important attempt to codify some of the ideas behind program environment design was Hutchins, Hollan and Norman's paper about direct manipulation and cognitive distance [25]. As they describe it, cognitive distance from a computer task is split into the aspects of semantic and articulatory distance.

Semantic distance is the difference between what the user intends to express and what they have to actually say to do it. For example, a user may wish to pass input data through a series of computations and then produce a graph. If they do this by drawing lines between boxes that represent the data, computations, and the graph, this would represent a small semantic difference because the user's actions directly indicate what they wish to be done. If they instead perform this task by writing hundreds of lines of code to manipulate the data and manage passing the results around, the semantic distance would be large. A larger semantic difference does not mean a task cannot be achieved in a particular environment; it simply means that the user is not able to directly specify what they wish to accomplish. Having a small semantic distance is important because it allows the user to focus on their objective instead of on the mechanics of making the system work.

Articulatory distance is the relationship between the physical or on-screen representation of an expression and its meaning. For example, a picture of a square would have a smaller articulatory distance than a list of line segments or corner coordinates if the task was to draw a square. Nevertheless, a picture is not always automatically the best representation. If the task is dictation, the word "English"

would have smaller articulatory distance as a representation than the picture in Figure 2.1. Note that the computer's internal representation is not of concern for this measure; the waveform may very well be the computer's desired representation for the same dictation task.

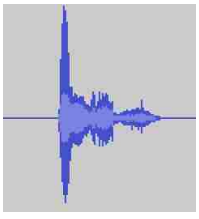

Figure 2.1: A waveform of the word "English"

A small articulatory distance is desirable because it allows the user to rapidly verify that their commands are being interpreted correctly and that their desired actions are being taken. With a larger articulatory distance, the user must spend more time mentally mapping their internal model to the inputs expected and outputs produced by the computer.

This idea was further explored by Green and Thomas in their "match-mismatch" theorem [22, 23], where they state that different representations are more productive for different tasks. For example, they empirically found that a declarative, Prolog-style language was superior for performing "backwards" reasoning from given outputs to determine plausible inputs, whereas an imperative-style chain of nested conditionals was superior for reasoning from given inputs to expected outputs. They found that this difference in efficiency applied regardless of the use of visual or textual programming languages, and that textual languages in fact outperformed visual languages across all the tasks used in their studies [33].

Chapter 3

Theory

The previous chapter provided background information about spoken programming in general, as well as previous research in the area. With this fresh in mind, we turn our attention in this chapter to what this specific dissertation covers and how it differs from previous research. We provide a high-level description of our new programming language, together with its syntax and features. We then describe the programming environment and how it incorporates specific types of context to influence the programming process. Details of how we implemented the items described in this chapter can be found in Chapter 4.

3.1 Rationale

The first question after deciding on spoken programming is this: What language should we speak? An initial thought is that we should speak something like Java or C, since those are such popular languages. However, as discussed in section 2.1.2, attempting to speak preexisting computer languages has been problematic. Rather than a deficiency in the efforts of previous researchers, we believe this is a deeper problem with the whole idea of attempting to speak a traditional computer language.

First, languages like Java are full of meaningful punctuation and other typically non-speakable symbols. For example, in the statement

```
x := 5;
```

there are only two symbols that are part of normal spoken English (x and 5). Depending on their programming background, native language, English dialect, etc., different speakers might easily come up with different spoken variants of this statement, such as:

- x colon equals 5
- x equals 5
- set x to 5
- x gets 5
- assign 5 to x
- store 5 in x

In fact, using an off-the-shelf commercial speech recognition tool to try to dictate the line above, one must actually say

```
letter x space bar colon equal sign 5 semicolon new line
```

A larger example of the difficulty of writing programs in traditional languages with speech recognizers designed for English is found in section 6.2.

Previous spoken programming research has largely attempted to do exactly what was described above, namely to generate code in traditional languages by attempting to generate normal textual language statements from some sort of spoken syntax,

either reverse engineered from the written syntax or via some entirely new syntax. Regardless of the exact spoken syntax chosen, the written language itself is never modified. Our contention is that if spoken input is an important consideration, we should instead be modifying the written language to suit speech.

Our language syntax is built from statements that use small English phrases and a restricted subset of English sentence structure. To the largest extent possible, anything that is not normally a part of spoken English has been omitted or minimized. Identifiers and keywords are case-insensitive, since word case is not normally spoken. Spacing and indentation are allowed as "secondary notation" [33], but are not meaningful as components of the syntax. Control flow, block structure, and other items that might traditionally be indicated with difficult-to-pronounce punctuation are either explicitly spoken or are omitted entirely. For example, to set a variable x to the value 5, the syntax is "`set x to 5`". A few exceptions have been made for a few well-known arithmetic symbols (e.g., '+' instead of "plus", '<' instead of "less than", '=' instead of "equals").

This is all in keeping with the ideas of minimizing both semantic and articulatory distance. In terms of semantic distance, the user's intention of producing code is nearly directly realized, i.e., the sentence they need to speak to produce a particular line of code is essentially the exact sequence of words in that line of code. It is hard to imagine anything more direct than that. On a slightly deeper level, the language we provide is fairly low-level, so one might argue that there is a larger semantic distance between the program the user wishes to write and the lines of code they must create. This argument has merit, but the situation is no different for a person who wishes to write a program in C. Since our language is higher level than C, the semantic distance should be no greater.

In terms of articulatory distance, one might argue that the form of a word on the screen doesn't really have much to do with the actual sound or meaning of that word, but this barrier has already been overcome when the person forms those

associations in the process of learning to read and write. Thus, the on-screen textual representation as English words and phrases should also present minimal articulatory distance.

3.2 Language Design

As mentioned, our language syntax is intended to evoke the feeling of speaking English. While not being fully general English sentences, the individual statements are built from phrases that are either grammatical—though restricted—English phrases or are at least plausible chunks that would be spoken by an actual person. The specific syntax of each construct was determined in a fairly ad hoc manner by informal surveys and discussions around the UNM Computer Science department. Most constructs had a single dominant popular choice of phrasing. In the cases where there were alternatives with no clear winner, we chose one based on what felt most natural to ourselves. While certainly not a rigorously formal syntax development process, we feel that the end result is natural enough to be easily learnable and comfortably speakable by a fluent English speaker. The truth of this claim will be explored in Chapter 6.

The most popular languages today (Java, C, Python, etc.) are all imperative languages. The functional and logic programming paradigms offer some appealing benefits for the design of a language, but the use of these would require many potential users to learn both a new spoken syntax and a new programming paradigm at once. This would have made it difficult to design a study that tests the effectiveness of the syntax unless we limited testers to people who already had existing functional programming experience. We felt that this restriction would have unduly restricted our ability to run the study described in Section 5.2. Therefore, to maximize accessibility of the language, we chose to make it an imperative language.

Even in the imperative paradigm, there is a wide variety of capabilities provided by the programming language, ranging from assembly to modern high-level object-oriented languages. Because the purpose of this research is not programming language features per se, we have chosen to omit many of the more complex features. The language we have built provides similar functionality to C in terms of control structures and data types, but we have incorporated some higher-level features such as automatic memory management and type inference. The individual features will be described in the sections below.

Also like C, our language is statically typed. While the common trend among recent languages seems to be in the direction of dynamic typing (consider Javascript, Python, etc.), the use of dynamic typing discards a significant amount of contextual information that can be used prior to runtime. Because the use of extra context to improve programming performance is one of the key ideas in this research, we did not wish to sacrifice this source of information; thus, static typing.

However, just because the language is statically typed does not mean we must force the programmer to litter their code with type declarations. Instead, we use type inference [34] to automatically recover this information at compile time (or even at program dictation time). This allows the programmer to focus on their algorithms rather than on the specifics of variable types, while still allowing us to make use of type information for speech recognition purposes.

While the combination of features above do not make an interpreter impossible, they do mean that the entire source file must be processed to infer types and resolve forward references before execution can begin. Because the design of runtime language environments is also outside the scope of this research, we found it more expedient to simply generate an executable after the initial processing rather than create an additional runtime interpreter.

The overall combination of the decisions above results in a minimalist, C-like lan-

guage for spoken programming. Though minimal, our language is Turing complete. A proof of this can be found in Appendex C. Nevertheless, it is not meant to be a "production-ready" language in the sense of being usable to build large, real-world systems. Many features that are considered indispensable for modern software engineering are missing, but this is not a problem for the purposes of investigating spoken programming.

Next, we describe the individual language features and their syntax. The following gives an intuitive description of the syntax and the reasons for choosing it. For a formal grammar of the complete language, see Appendix A.

3.2.1 Variables and Data

The core of any imperative language is state manipulation, and the standard way to provide this is through variables and variable assignments. In order to assign variables, it is also necessary to have a syntax for literals of each of the supported types. It seems necessary to support string and integer variables in order to perform any non-trivial tasks. Booleans can be trivially emulated using integers, so they are not included as a separate type. Similarly, characters can be treated as a special case of strings. We also provide arrays of the above, but more sophisticated combinations such as sum types (variants), product types (records/structures), and user-defined types are all omitted.

The basic statement to create and update variables is called `set`. It takes an expressions and stores it into the named variable or array reference. If the variable did not previously exist, this creates the symbol and assigns its initial type. If the variable already existed, the type of the new value must be compatible with the previous type.

Examples:

```
set n to 1
set y to the string Hello
set element n of x to y
```

Note that string literals are introduced by the words "`the string`." The string value extends from there to the end of the line. When speaking, this is indicated by pausing at the end of the string. This is one of the few concessions to the use of non-visible features to indicate meaning. This decision was necessitated by practicality; further details will be provided in section 4.3.

In order to perform computations with its state, the language needs to include statements that can combine and manipulate variables. For numeric variables, this consists of arithmetic expressions. We have implemented addition, subtraction, multiplication, and division. More complicated expressions can be built using standard order of operations to enforce precedence.

Sensible basic manipulations for string variables are concatenation, substring extraction, and simple search (like the *strstr* function in C). Other string functionality can be easily built on top of these basic operations, so additional functionality has not been built in.

Modern languages also typically provide automatic memory management. In languages with pointers and references, this is often provided through garbage collection or reference counting. In this language, there are no pointers and no references, so memory for scalars can be automatically allocated and deallocated based on the call stack at runtime. Arrays with dynamic length can require dynamic memory allocation at any time, but this is handled automatically by a small runtime support library we have written. Since the array variables cannot outlive their scope, they do not introduce any additional difficulties with deallocation. Thus, our language does not need to provide any explicit memory management facilities.

Examples:

```
Set x to 5
Set z to x + 1
Set element z of y to z + 5 * x
```

3.2.2 Functions

Since Dijkstra's famous letter on structured programming [13], it has been considered inconceivable to program in any language that did not include structured programming facilities. The most important such constructs are functions and loops.

Our language supports definition of simple functions that accept zero or more positional parameters with the types described above. Functions return single values, again of any of the supported variable types. Functions may also return nothing (the equivalent of *void* in C). More advanced parameter schemes like optional parameters, default values, and named parameters are not present. Because the language does not include pointers, all function parameters are passed and returned by value.

Examples:

```
define function main taking no arguments as
    ...
end function

define function foo taking arguments x as
    return x + 1
end function

define function bar taking arguments m and n as
    ...
```

```
end function

define function baz taking arguments m and n and z as
    ...
end function
```

There are two small oddities to notice here: the word `arguments` is always plural, and argument lists are joined with `and` instead of commas. This is intended both to keep the syntax regular and to avoid non-spoken content. While "`taking arguments x`" may be ungrammatical, we have found that it causes no problems for users; additionally, the speech recognizer can automatically correct if the person says "`taking argument x`," so there is no impediment to users' potential word choice here. The unusual list structure is slightly more disconcerting for people, but the alternatives were to explicitly say "comma" between words (which would be at cross-purposes to our design goals) or to use pauses to distinguish between identifiers (which was neither robust nor easy for users to discover).

Unlike variable symbols, which are created at their time of first assignment, all functions are conceptually in scope from the first line of code. Naturally, this requires the compiler to make multiple passes over its internal abstract syntax tree (AST), but this is not uncommon. More importantly, it enables forward references to functions (but not variables) that have not yet been defined at the point of their reference, and it enables recursion. Functions may call themselves recursively, and they may be mutually recursive through some chain of intermediate functions.

Like variables, function parameter types and return types are determined via type inference. However, functions are not polymorphic; a particular function's types must resolve to a single type solution. This limitation is not inherent in the language grammar, but is simply an implementation limitation caused by the underlying code that the compiler generates. This limitation could be lifted without modifying the

language syntax, but because polymorphism itself is not part of this dissertation, we have not done so.

In addition to defining functions, a programmer must be able to call them. In order to keep the natural English feel of the language, we found it necessary to provide two syntaxes for this. For functions where the result is thrown away (called for purposes of side effects), the syntax is `call` *`function`*. If the function takes arguments, one must append `with` *`arguments`*. For function calls where the result will be stored in a variable or used in an expression, the syntax is `the result of calling` *`function`*. Arguments are appended the same way as the `call` statement.

Examples:

```
call f
call f with 1
call f with 2 and x
set x to the result of calling f
set x to 5 + the result of calling factorial with 10
```

From a language design standpoint, it seems mathematically inelegant to offer two syntaxes for the same purpose (of calling functions). However, the alternative requires either verbal gymnastics or the creation of unnatural sounding statements such as "set x to 5 + call f". Does that statement mean "set x to 5, plus (also) call f", or does it mean "call f, add 5 to the result, and store that value in x?" We know the grammar restrictions that it means the second, but our casual daily use of "plus" to mean "also" can lead to the wrong intuition. Rather than either of these alternatives, we provide the two separate function call syntaxes.

3.2.3 Looping

There are two basic kinds of loops: definite and indefinite. In a definite loop, the loop body will execute some fixed number of times, while an indefinite loop is repeated an arbitrary number of times as long as its guard condition remains true. It is simple to simulate a definite loop using an indefinite loop plus a counter variable, but it is impossible to simulate an indefinite loop using a definite loop. Therefore, the language includes syntax for indefinite loops and omits definite loops.

The syntax for indefinite loops is the very typical `while` *`condition`* `do`. Exactly as one would expect, the condition is evaluated each time the `while` statement is reached, and the body will be executed repeatedly as long as it remains true.

Examples:

```
while 1 = 1 do
    ...
end while

while x < 5 do
    ...
end while
```

The body of the loop is a nested scope. It inherits the symbols visible in the parent scope, but any new variables created within the loop body will not be visible beyond the execution of the loop.

3.2.4 Conditionals

In theory, it is possible to simulate conditional statements (*if-then-else*) using a combination of indefinite loops and extra guard variables. The same reasoning that

justifies omitting definite loops could also justify omitting conditionals. However, conditionals are so common, and the emulation is so cumbersome in comparison, that this seems unreasonable. Therefore, in the interests of avoiding the Turing tar pit [32], we have also included a built-in syntax for conditional statements. Like many functional languages, our `if-then` always includes an `else` and an explicit terminating `end if`. The primary purpose of this decision is simply consistency, although it does also have the benefit of avoiding the "dangling-else" problem.

Because this may then result in unwanted `else` blocks, we also provide the statement `nothing`, which allows the user to state that nothing should happen. This statement is primarily intended for use in otherwise-empty `else` blocks, but it is valid anywhere a statement can go.

Examples:

```
if x < 5 then
    ...
else
    ...
end if

if x > 1 and x < 10 then
    ...
else
    nothing
end if

if not z = y or y >= 0 then
    nothing
else
    ...
```

```
end if
```

Like the body of the `while` loop, both the true and false branches of the conditional introduce new nested scopes.

3.2.5 Booleans

Although we declared in section 3.2.1 that Boolean variables were unnecessary due to the inclusion of integers, both conditionals and indefinite loops require Boolean tests. Thus, simple Boolean tests and logic must be provided. These values can be used in `while` conditions and `if` statements, but cannot be directly stored into variables or otherwise treated as first-class values.

For all variables, we include tests for equality via the normal equals sign (=). For numeric variables, we also permit numeric comparisons (<, >, <=, >=). In order to complete Boolean logic, we will also need to be able to combine these with `AND`, `OR`, and `NOT`.

Examples can be found in the previous two sections.

3.2.6 Input and Output

Any program that cannot perform input and output is only useful for turning your CPU into a space heater. Thus, we have naturally provided some basic I/O facilities. The `print` statement evaluates an expression and prints the result to standard output. It does not produce a new line after its output. We provide the `new line` statement to explicitly control line breaks.

Examples:

```
print 5
```

```
print x * 2
print element 5 of a
new line
print the string Hello
```

The `read` statement reads a value from the user and stores it in the named variable. Like the `set` statement, this automatically creates the variable if it did not previously exist within the current scope. The type of the variable is automatically determined by the type inference procedure based on other uses of the variable, and the `read` statement automatically coerces the input into the correct type.

Examples:

```
read x
read element 1 of y
```

All modern operating systems provide support for redirection of a program's standard input and output to and from files. Therefore, it is not necessary to provide explicit support for file I/O. Network sockets, shared memory, and other interprocess communication facilities are also omitted.

Most languages support some kind of external library linkage. This is a vital feature that would need to be present in any programming language made for serious use, but it is not necessary to demonstrate the viability of spoken programming. In addition, interfacing to external libraries written in other languages re-introduces the problem of trying to create a spoken syntax for other programming languages. Therefore, this feature has been omitted, but this will be an important area for future study.

3.3 Making Use of Context

Given the spoken programming language described above, one might expect that a regular, off-the-shelf English recognizer would perform significantly better than it would when trying to dictate C or Java. But can we further improve the programming experience by incorporating additional context information? Generalized speech recognition is all about context; the most successful products typically work in limited domains where the user's vocabulary can be interpreted within a specific task-oriented frame of reference. Even commercial "general English" speech recognition packages are often specialized for special domains such as legal or medical dictation [15].

Similarly, when writing a program, we don't just have the text coming out of the recognizer. We also have many additional types of context induced by both the program text and the process of writing the program. This research shows that recognition accuracy can be improved in the programming domain by making use of these additional programming-specific contexts. Because context is an open-ended concept, this research will focus on three specific areas: Language grammar, identifier scope and type inference. In addition to providing a starting point for future research, this will serve to confirm that the lessons learned in decades of research in general speech recognition still apply in the development of a spoken programming environment.

In the following sections, we provide a high-level description of the three areas of context we are considering and how they can be used in a spoken programming environment. Specific details of exactly how we have implemented these ideas are found in Chapter 4.

3.3.1 Language Grammar

The first—and arguably most obvious—source of context is the language grammar itself. Chomsky's claims notwithstanding, most speech recognition software today has moved away from attempting to find a generative grammar for English and into the realm of probabilistic models. Moreover, people typically communicate with one another using incomplete sentences and other non-grammatical utterances. This presents no barrier to our day-to-day conversations, but even in this environment the speakers typically share a context that they use to interpret each other's utterances. For example, in the context of reading this document, the sentence "I like squirrels" would be a non sequitur that would be a distraction at best, or an impediment to understanding the surrounding material at worst.

Similarly, when writing a program, the programmer and programming environment immediately share the context of attempting to create a program. Utterances by the user should be interpreted not as random bits of English, but as attempts to either create a program statement or perform an editing command. Unlike English, a computer language does have a complete grammar of all valid sentences in the language. We can use this knowledge to significantly restrict what we expect the user to say, but in fact we can do even better: Because a program has not just a set of valid statements, but an entire structure, we can use the partially created program's context to guide the recognizer's decision about which statements are the most plausible. For example, consider the following partially dictated program:

```
define function f taking arguments x as
    while x < 10 do
        ...
```

If the user says "end *garble*" at this point in the program, what kind of a statement are they trying to dictate? It could either be `end while` or `end function`. If we insert `end function`, we produce an invalid program because the `while` loop is

not correctly terminated. Similarly, the user probably did not say `end if` because it doesn't make sense (and isn't valid) to try to terminate an `if-then` block when we aren't inside one. Thus, we can deduce that the user probably said `end while` and bias the recognizer toward that expectation.

3.3.2 Identifier Scoping

Because the symbol scoping in our language follows the nested block structure, different symbols are visible at different points in the program. We make use of this symbol scope information to guide the recognizer. At the basic level, by knowing which symbols are defined in any particular scope, we can ensure that the user doesn't refer to symbols that are not visible, and we can even do a limited amount of correction for mispronunciations or similar minor disfluencies.

Simply including or excluding symbols from the grammar based on the scope is a fairly blunt tool. On a more nuanced level, the principle of locality suggests that variable references that are near together in the code are more likely to refer to the same variable than references that are far apart. Many existing editors use scoping with auto-completion to suggest variable names, functions, etc., that are in scope when the user types the first few characters of the name. Using the scope to make more closely-defined symbols appear closer to the top of the list of alternatives often saves time.

In spoken programming, traditional auto-completion is not needed, because the programmer will be inclined to speak full words. Having to stop and spell out the first few symbols of an identifier would be a net time loss over simply speaking it out even if the auto-completion always guessed the correct symbol. However, locality of reference still applies. Given a probabilistic language model that most speech recognizers implement, the extension of auto-completion to voice input is then conceptually straightforward: Instead of auto-completing symbols based on

the first few characters typed, a vocal programming system should use scoping to automatically raise the expected probability of more closely scoped symbols and lower the probability of more remote symbols. This will reduce the number of recognition errors and improve accuracy. A number of prior researchers have made use of scoping to keep a simple list of variables in scope for the voice recognizer [10, 29], but they do not appear to have used the information in the more sophisticated manner examined here.

3.3.3 Type Information

Our final use of context is the type system. To save the programmer from the burden of declaring the types of all their variables, many modern languages are either dynamically typed or make use of type inference. Dynamic typing is powerful and easy for the programmer, but prevents the editor from knowing anything about the types before runtime. With type inference, on the other hand, the programmer is still free to use variables without worrying about type signatures, but the compiler is still able to perform compile-time validation. More importantly for the purposes of this project, the editor can also perform type inference to gain additional context information about symbols in the program. This additional context can be used similarly to scoping to enhance the selection of spoken symbols.

Based on type information, we can also determine if a particular symbol represents a function (valid for calling), an array (valid for extracting elements), an integer (valid for array indexes and arithmetic), or a string (valid for concatenation). We can determine the arity of a function (functions must be called with the correct number of arguments), and we can determine function argument types and return types. Other uses of the type system can easily be imagined, but have not been considered for this research.

For example, suppose that functions “flop” and “flap” are both in scope. Without

further context, it will be difficult to distinguish between these two functions when spoken. If "flop" is known to take an integer and "flap" is known to take a string, then the editor can immediately improve its accuracy by entering the correct function based on which type of variable the user passes as a parameter. Similar choices can be made when the user passes a variable into a function with a known signature, sets a variable to the result of a function with a known return type, etc.

3.4 Programming Environment

The programming environment acts as the glue that holds all the other pieces together. During dictation, it provides the visible UI for the user, accepts proposed program fragments from the speech recognizer, and updates the recognizer based on the context the user is creating. It runs the compiler when needed and handles saving and opening files. A schematic of the overall system design is shown in Figure 3.1.

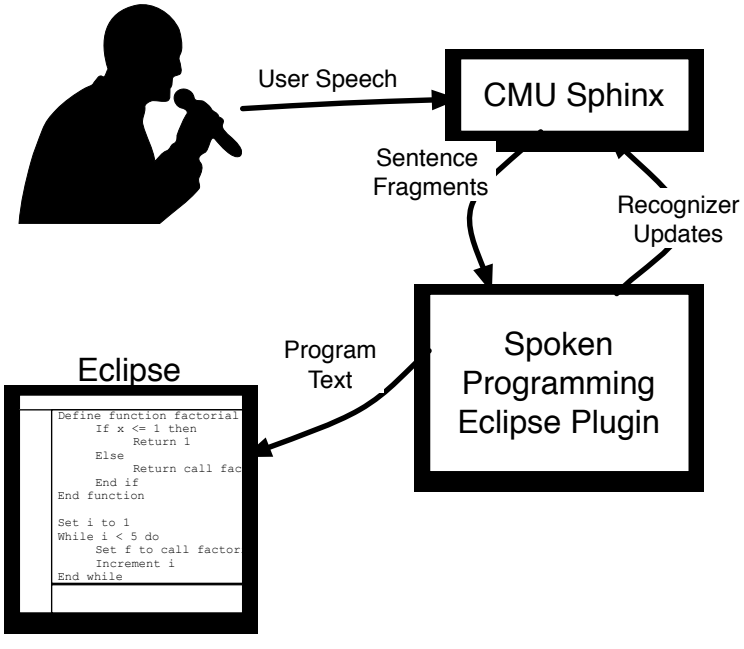

Figure 3.1: Design of the Spoken Programming Environment

Ultimately, the programming environment must also address other tasks in the programming workflow such as editing, source code navigation, and debugging. For the purposes of this research, we have kept these additions as simple as possible. We have ignored debugging entirely. Existing Java debuggers can be used to do debugging of our generated code, but a full spoken debugger is left to future research. Similarly, since we are studying initial program dictation, we provide no text navigation commands. Other researchers have studied spoken source code navigation [2, 29, 40], and their techniques could be combined with our system without dramatically changing either one.

Finally, we do provide editing, but only a single command: "fix that" or "no" causes the environment to back up a line and allow the user to try again. Though simple, this command provides everything necessary for interactive program dictation. More advanced editing facilities would be necessary for post-dictation corrections, but that topic seems best addressed in combination with spoken code navigation rather than in the initial prototype. In any case, more advanced editing commands would not have changed the design of our user study.

Chapter 4

Implementation

In Chapter 3, we described the rationale and design of our new language, as well as the programming environment and the context-based features it incorporates. In this chapter, we provide more specific details of how we implemented the ideas previously described.

4.1 Compiler

Once the language syntax has been established, the first component necessary to make a language do something is a compiler. Our compiler is a pretty standard pipeline, starting with lexing and parsing into an abstract syntax tree (AST). We then make several passes over the AST to resolve references and perform type inference. Finally, we output generated code. The trivial program in Figure 4.1 will form an example that we use throughout this section.

The entire compiler is written in Java, using the ANTLR parser generator [1] for all lexing and parsing tasks. In the first pass, we convert the input file into tokens and parse the token stream into an AST. With ANTLR, the form of the AST

```
1   define function main taking no arguments as
2       read x
3       set y to x + 5
4       print y
5   end function
```

Figure 4.1: Example Program for Chapter 4

is disconnected from the original source code, using generic node types instead of tokens from the source code. This pass ensures the correctness of the syntax but does not perform variable resolution or type inference. The AST for the program in Figure 4.1 is shown in a Lisp-like format in Figure 4.2.

```
 1  (FUNCTION main
 2      (BLOCK
 3          (read x)
 4          (set y
 5              (EXPR +
 6                  (EXPR x) (EXPR 5)
 7              )
 8          )
 9          (print
10              (EXPR y)
11          )
12      )
13  )
```

Figure 4.2: AST generated from program in Figure 4.1

The second pass performs initial variable resolution. It fills out the symbol table with both function and variable references. All variables can be resolved at this time because we require that variables be set to an initial value (with `set` or `read`) before they are otherwise used. Function calls, however, can contain forward references to functions that are not yet defined at the point in the source code where the call occurs. When this happens, we leave a placeholder reference in the symbol table that

is resolved later. Function calls that refer to functions that are already defined get resolved immediately. After this pass, all variable references and backwards function references have been resolved. In this pass, we catch any use of undefined symbols, variables that are read before being written, etc.

After processing the AST in Figure 4.2, the two x references will point to the symbol table entry created at line three, and the two y entries will point to the entry created at line 4. The *main* reference will point to the entry created at line 1. Table 4.1 shows what would be represented, with nested scopes shown separated by pairs of colons (`::`). Note that the entries for the same symbol actually point to the same symbol table entry even though they show up on two different lines.

Line	Symbol	Type	Definition
1	*main*	unknown	1
3	*main::x*	unknown	3
4	*main::y*	unknown	4
6	*main::x*	unknown	3
10	*main::y*	unknown	4

Table 4.1: Symbol Table After Pass 2

The third pass performs both type inference and symbol resolution for the leftover function call forward references from the second pass. Our type inference algorithm follows the scheme laid out by Damas and Milner [9]. As we process the AST, each node in the tree that requires a type generates a type constraint. These may be of the form typeof(*var*) = typeof(*expression*) or typeof(*expression*) = *type*. Once the full set of constraints is generated, we use a unification algorithm [30] to generate a solution and set the types of every symbol. If there are type clashes during unification, we stop and report the mismatch to the user.

In our example, there are no forward function references or undefined variables in this program, so this third pass does not resolve any additional symbols. The type constraints generated are shown in Table 4.2, along with the line that creates each

one. Note that the line numbers from the AST are shown in the order that the nodes are processed in the depth-first traversal, so they are out of order numerically.

Line	Constraints
1	typeof(*main*) = *function*; typeof(*main::args*) = *empty*
6	typeof(*5*) = *int*
5	typeof(*main::x*) = typeof(*5*); typeof(*main::x+5*) = int
4	typeof(*main::y*) = typeof(*main::x+5*)
12	typeof(*main::return*) = *empty*

Table 4.2: Type Constraints from Figure 4.2

After generating all the type constraints, we apply unification to solve them and find that typeof(*main::x*) = *int*, typeof(*main::y*) = *int*, and typeof(*main*) = *function*⟨*empty,empty*⟩. Notice that *main*'s type includes the fact that it is a function as well as its return type and argument types. We then update the entries in Table 4.1 to yield Table 4.3.

Line	Symbol	Type	Definition
1	*main*	*function*⟨*empty,empty*⟩	1
3	*main::x*	*int*	3
4	*main::y*	*int*	4
6	*main::x*	*int*	3
10	*main::y*	*int*	4

Table 4.3: Symbol Table After Type Inference

Despite the simplicity of the type system this language provides, it is still possible to fail to generate a unique solution to the type constraints. If there are untyped variables left at the end of unification, this is handled in many type inference systems by setting those variables to some type of generic type or "Object" superclass. Since we do not provide a generic type or any object hierarchy, we instead stop and report this as an error. This situation arises only if the symbols in question were not used as part of any computation that imputes a type. The program in Figure 4.1 did not have this problem, but the program in Figure 4.3 would fall into this situation. Since

the program can't do anything useful with such a symbol, we have not provided a generic type to handle the problem. Instead, if the user requires a symbol that is used this way, they need to provide a dummy calculation that uses the symbol to force its type (such as adding 0 to make it an integer or setting it to an empty string prior to reading it to force a string).

```
define function main taking no arguments as
    read x
    print x
end function
```

Figure 4.3: An Untypable Program

The final pass is code generation. We use the StringTemplate library that comes with ANTLR to produce Java code as output. To use StringTemplate, we walk the AST and generate a tree of nested templates that mirrors the AST. Each template takes parameters and produces strings as output, either directly or by referring to embedded child templates. Once the tree is built, the StringTemplate library combines all the parameters with the templates to produce the final output. The Java code generated by the user's program is combined with a small support library we wrote to handle tasks like array auto-extension and input type conversion, and the complete program is linked into a runnable Java class.

We generate Java as output primarily because the rest of the system is written in Java, and this keeps everything consistent. However, there is nothing fundamental about the use of Java; by swapping the StringTemplate templates and writing an appropriate support library, this compiler should be able to support any modern language as output (or even assembly with enough effort put into the support library).

Since we create a Java program as output, an interesting future expansion would be to allow linking to the myriad existing Java libraries. However, since we did not provide spoken syntax for Java, we also omitted this ability from the compiler.

4.2 Programming Environment

A programming environment is not just a programming language; it also provides all the program creation and editing features that make a programmer productive. Our environment is provided in the form of an Eclipse [16] plugin that integrates CMU Sphinx [41] speech recognition with the existing Eclipse editing capabilities and the compiler described above. Because Eclipse and Sphinx are both written in Java, our plugin is also entirely written in Java. Details of each of these will be provided below.

A common design feature of a voice recognition system is a probabilistic language model. The language grammar describes *possible* sentences, and the probabilistic model adds a measure of *likelihood* of sentences. For example, the phrase "go to town" is much more likely to be found in an English sentence than the phrase "go to towel." To create a model that can represent these probabilities, large corpora of speech or written text are divided into n-grams and the resulting co-occurrence frequencies are analyzed [27]. For maximum effectiveness, the corpora are preferably recorded in the same domain anticipated for subsequent speech. However, since part of our goal is to reuse existing English speech recognition, we have not developed a separate spoken programming corpus. We use the built-in English models provided with Sphinx.

In the CMU Sphinx system, there are three types of models. At the lowest level, the acoustic model allows the system to turn raw audio data into a sequence of phones. Next, the phonetic dictionary provides mappings of words onto phones. This allows the recognizer to produce lists of possible words based on the phone sequence. At the highest level, the language model restricts which combinations of words can be produced. The language model is not an absolute listing of possibilities, but is a probability distribution learned from training on appropriate corpora.

In programming, a high-level English-trained language model is of limited use, because many common programming idioms will not correspond to normal conversa-

tional English. We reuse the two lower-level models unmodified, and instead replace the top-level models with grammars derived from our spoken language syntax as described in 3.2.

4.2.1 Grammars

Sphinx accepts grammars written in the JSpeech Grammar Format (JSGF) [26]. A JSGF file contains a grammar in a format similar to a BNF grammar, but with some extensions for probabilistic recognition and tagging. The initial grammar is loaded from disk, but can be subsequently modified or swapped at runtime.

Our use of grammars has gone through some significant evolution throughout the process of building the environment. Initially, we simply took our ANTLR grammar and converted it to JSGF. While this produced accurate results, it had an important shortcoming: Because Sphinx attempts to process entire "sentences" from the grammar at once, it required the user to speak their entire program in a single breath without pauses! This makes a nice trick to show off at parties, but is obviously unusable from a programming point of view.

Our second attempt was to break down the grammar structure into small phrases and snippets that could be comfortably and logically spoken between pauses. This idea is similar to n-grams, but instead of using fixed-size chucks of words, we split the grammar at approximately statement boundaries. For compound statements, we split `while` *`condition`* `do` and `end while` into separate statements for grammar purposes, but simple statements like `print` and `set` *`variable`* `to` *`expression`* are recognized as a single chunk. Each of the sample programs presented throughout this dissertation is split into lines at the same boundaries where we have divided the grammar chunks.

Compared to the first attempt, it was now easy to speak individual statements,

but nearly impossible to dictate a complete program. Without the context imposed by the original grammatical structure, the recognizer could (and did) insert statements in any order. Instead of mistaking a symbol name or a literal value, it would mistake an entire construct for another. The accuracy was so low as to be unusable.

The final attempt came about when we hit on the idea that programming can be treated as a dialog. Like our second attempt, individual sentences at any particular time are relatively freeform and interchangeable. But like our first attempt, the ongoing context of the conversation imposes an overall structure and circumscribes the sentences that are relevant. Certain transition words and phrases alert the participants that the context is being adjusted or changed. Like a human dialog, the participants take turns, and the context tells them if the other participant's most recent utterance makes sense. Of course, there are also differences: An out-of-context utterance in a human conversation is generally a non sequitur that can be recovered after the other person "figures out" what happened, whereas an out-of-context utterance in a program is simply an error.

To implement this dialog idea, we split our second grammar into multiple grammars. Each smaller grammar contains only the chunks that make sense within a particular programming context. For example, the only construct allowed at the top level of the program is function definition, so the top-level grammar only contains chunks related to defining a function. Once a function definition has been completed, that acts as the transition phrase that tells the system to push the context onto a stack and switch to the function body grammar. In the function body, all chunks that introduce a legal statement are part of the grammar. Statements that introduce further nested blocks and nested scopes cause new grammars to be loaded as the context switches again. For example, the `end while` statement only makes sense inside the "while body" context because an `end while` that comes before the corresponding `while-do` would never be part of a correct program. Other statements that introduce new blocks (and new grammars) are `if-then`, `else` and strings. The

reason why strings require a separate grammar will be described further below. We make use of the JSGF tagging feature to enable these transitions whenever a certain grammar rule is triggered without looking for specific words in the source code.

When a block comes to an end, we need to return to the previous context. Since the structure of nested contexts largely mirrors the nested block structure, we can use the same stack structure to pop out and return to the previous context whenever an "out" transition is called for. However, allowing editing puts a large wrinkle into this concept. For example, if the user says `end function` then we would pop the function context they are in and return to the global context and its associated grammar. If the next thing they say is "no" or "fix that" (our editing command), the environment must back up a line and allow the user to try again. This requires us to "un-pop" the stack and restore the function context. Since we dynamically modify the grammars as the user is programming, it is not even sufficient to reload the previous grammar. Instead, we attach a pointer from every line of source code to a sort of closure that represents the full state of the stack at that line. When the user wants to make a correction, we can restore the stack directly from the code at that point where the correction is being made.

4.2.2 Scoping

To implement the identifier scoping we described in Section 3.3.2, we dynamically modify the grammar rules as the user is dictating. The grammar contains rules for how to form legal identifiers, but rather than a single rule for an identifier, places that make different use of the symbols have different rules. For example, expressions refer to existing variables; they cannot create new variables. Thus, we have a rule for *definedVars* that encapsulates all the known variables. A `set` statement, on the other hand, potentially creates a new variable, so it will refer to a rule called *possibleVars*. In addition to the known variables, *possibleVars* also allows the introduction of new

variables.

Each time a new block is entered, a fresh copy of the base grammar for that appropriate context is created. The new grammar is then modified so that *definedVars* and *possibleVars* contain references to all the defined that are visible from the new scope. This list is built by walking up the scope tree and getting a list of all symbols defined at that level. The symbols at that level are appended to the partial list created at the lower levels. For the case of *definedVars*, this process stops once the top-level scope is reached. For the case of *possibleVars*, the rule to match a generic identifier is then appended to the end of the list.

The description above explains how pure scope visibility can be implemented, but we can actually be a little more sophisticated. JSGF allows each rule to have a weight. The weights are not true probabilities, but they do factor into the probabilities Sphinx uses when it turns the grammar into a speech model. Instead of purely appending each symbol list as described above, our implementation starts with a fixed weight and reduces it by a small fraction for each level it moves up the tree. When appending the symbols, they have whatever weight is calculated for their level attached. The optimal weight adjustment value needs to be determined empirically, but because our test programs turned out to be too small to see any large changes from varying the weights, we have not attempted to find an optimum. We have left this as an area for further research once a larger user study is performed.

Each time a statement is dictated that may modify the set of variables in scope, the algorithm above is executed to update the current grammar. This includes `set`, `read`, and entrance to a new scope. When a scope is exited, we pop the current grammar as part of the process described in the previous section, so it is not necessary to explicitly update the grammar rules. The context for our dialog grows steadily by the introduction of new symbols, and then shrinks back down as the user stops talking about a particular block and moves on to the next one.

4.2.3 Type Inference

The type inference context feature is built on top of the scoping described in the previous section (you have to know what is in scope before you can attempt to give it a type). Much of it works conceptually the same, but the details are enhanced to add type information. Instead of a single *definedVars* rule, we have *definedInts*, *definedStrings*, etc. Functions are tracked at the grammar level based only on arity, so we have rules for *function0args*, *function1arg*, etc. The *possibleVars* rule is split into *possibleInts*, *possibleArrays*, etc.

Our full type inference algorithm is not practical for generating these rules at runtime for several reasons. First, the algorithm expects to be run on a complete AST from a syntactically correct program; it does not have provisions for dealing with undefined variables, incomplete functions, or other similar partial or incorrect constructs that occur in the process of creating a program. Second, the full type inference algorithm is not incremental; it expects to generate and solve all the type constraints for a complete program at once rather than add them individually as lines are added. Last, the algorithm is a little too slow to be used for realtime type inference without introducing user-visible delays. This is an implementation issue rather than a fundamental computational complexity issue, but taken in combination with the first two reasons, it makes the full algorithm unsuitable for the programming environment runtime.

Instead, we wrote a restricted version of the type inference algorithm that uses pattern matching to detect when variables have a definite type. For example, defining a function with three arguments is easily matched by a regular expression and can't be confused with a function with two arguments or four arguments. Similarly, statements such as `set X to 5` or `set element X of A to 5` strongly indicate that X is an integer and A is an array. Anything that doesn't match one of the "easy" cases caught by the pattern matching is left as an unknown type.

When generating the lists for all of the new rules, we walk up the scope chain just as described in the implementation of the scoping context. However, instead of fetching a list of all symbols at each scope, we only fetch a list of all integers or all strings or whichever type we are building the rule for. In addition, we must account for the symbols with unknown types. Because we don't know what type these variables are, we include them as candidates for every rule. The statement `read X` creates a variable X that shows up in *definedInts*, *definedStrings*, *possibleInts*, etc., until the user dictates a statement that constraints the type of X. Once that occurs, the rules get rebuilt and X appears only in the rules that match its newly inferred type.

4.2.4 String Handling

Strings are a bit different from other grammatical constructs. Unlike statements in the program, strings can generally contain anything from single characters and words to full English prose. Sixty years of AI research has established that grammars are not well-suited for recognizing this kind of unstructured chaos, but the rest of our implementation is built on top of JSGF grammars. How can these be reconciled? It turns out that when we swap grammars in Sphinx as described above, we are actually swapping the entire recognizer engine. It is entirely possible to load an entire n-gram model or other more advanced recognizer and swap it into place when a string needs to be input.

For the purposes of this dissertation, we have created our string recognizer as another simple rule-based grammar that contains a couple dozen words and phrases that were used throughout our sample programs. A more production-ready version of this environment could replace our trivial recognizer with whatever level of sophistication it needed, as described above.

The compiler has a simpler time. Since it takes text that has already been

translated from speech, it only needs to be able to detect the beginning and end of a string. We have handled this by making it accept a string as any sequence of characters from the beginning of the string (indicated by `the string` in the syntax) through the end of the line. While this does add one instance of whitespace sensitivity to the language, it seems better than the alternative of requiring the user to say "end string" or similar terminating commands.

4.3 Practical Considerations

A few of our implementation decisions were driven by software limitations or practical considerations. We describe these here.

First, we have attempted to avoid the use of prosody or other linguistic concepts that can't be directly translated into text. An early proposal suggested that prosody might provide a valuable additional source of context for programming. Instead, we found that users's pauses and speech speeds were too unpredictable to be used for anything. We made one necessary concession to this: When the user pauses for about half a second, Sphinx takes their audio up to that point and tries to recognize it as a sentence in the grammar. We used this by breaking our grammar into chunks that were small enough to speak without pauses, as described in Section 4.2.1. We also incorporated this into the programming environment by breaking the program into lines at the places where the user pauses. Since the chunks correspond approximately to statements, this results in the program being generated line-by-line.

We have also attempted to minimize dependence on any non-visible features of the textual language, such as whitespace. Line breaks are inserted as described above, but programs can largely be compiled regardless of the location of line breaks or indentation. One important exception is in strings. As mentioned in Section 4.2.4, we require that strings be terminated with a new line. This has not proved to be

any hinderance or surprise to users in our study.

Although the compiler is case-insensitive, we only generate lowercase output from Sphinx. Some of the sample programs are written with mixed case to demonstrate that it works, but we felt that the effort to produce similar output was not relevant for the outcome of this research.

Chapter 5

Experiments

In Chapter 3, we described the design of our spoken programming language and environment. In Chapter 4, we described the implementation details of the same. In this chapter, we now explain how the environment was evaluated: first through a preliminary case study, but primarily through the user study described below.

5.1 Case Study: Comparison to Dictation of C

In an initial case study, we used a commercial speech recognition product (Dragon Dictate 3.0 for Mac [14]) to compare the experience of dictating a program in C to a comparable program in our new language. C was chosen instead of other, more modern languages because it came the closest to giving a one-to-one correspondence in lines of source code and functionality per line. The C program is here:

```
#include <stdio.h>
int main(int argc, char **argv) {
        int i = 1;
        while ( i <= 10 ) {
```

```
                printf("%d",i);
                printf("\n");
                i = i + 1;
        }
        return 0;
}
```

The corresponding program in our language is here:

```
define function main taking no arguments as
    set i to 1
    while i <= 10 do
        print i
        new line
        set i to i + 1
    end while
end function
```

Both programs print the numbers from 1 through 10 (inclusive), one number per line. The C program has an additional first line (the `#include`) that is required to make use of `printf`, and an additional `return` statement that is necessary due to the calling convention for `main`.

To perform this experiment, we first performed the initial training steps required by Dragon and worked through the tutorials to learn its syntax. We then experimented with the system for approximately an hour to learn how to pronounce and/or spell each punctuation symbol and non-English word. Dragon also has rules for automatic capitalization and spacing, so we learned how to incorporate the commands needed to override the rules where needed. Finally, we dictated the C program and our program. We tried each program three times and measured the number of corrections needed and the time needed to complete the program. The results reported

in Section 6.2 are the average of the three attempts. In each case, we were able to dictate the exact program text (up to non-meaningful differences in spacing), so we have not reported accuracy numbers.

As a second part of this experiment, we also compared the results of dictating the spoken program in Dragon to dictating the exact same program in the spoken environment. The results of this are reported in Section 6.3.

5.2 User Study Description

While the case study described above with Dragon provides some intuitive validation of the claim that our English-derived syntax is easier to recognize with a standard English-trained speech recognizer, it does not provide any data about the full programming environment. To investigate the full system, we performed a user study as described here.

We recruited subjects from around the university. To participate, the subjects had to meet the following criteria:

1. Native speaker of English
2. General American accent (similar to what one would expect to hear on a network news broadcast in the United States)
3. No known speech impediments
4. Completed several semesters of Computer Science coursework or equivalent professional experience

The purpose of criteria 1–3 was not to discriminate against people of other backgrounds, but because Sphinx is known to produce variable results depending on

users' accents and speech patterns. To eliminate this variable from the study as much as possible, we selected an accent that was known to work well and required all participants to have a similar accent. The purpose of criterion 4 was to ensure that difficulties with writing our sample programs would not stem from users' inexperience or discomfort with writing short programs.

We started with thirteen initial subjects, but one was removed because his accent differed from General American by too much for Sphinx to produce usable results. Of the twelve subjects who were included in the final results, ten were male and two were female. Two had prior experience with voice recognition software, but ten reported having never used it. Two participants were self-described as novice programmers, five as intermediate, and five as expert. Two participants were undergraduates, two were non-students, and eight were graduate students.

Prior to the study, we wrote ten sample program that exhibit all of the legal constructs in our language. We arranged them into a series that built up from trivial programs to modest programs that sort a list of numbers or compute Fibonacci series. The full text of the programs can be found in Appendix B.

Each subject came in for a one hour recording session. We divided each session into three sections. In the first section, the subjects read each program without access to the interactive system. They simply read from the paper without reference to anything that was happening on the screen. Each program was read three times: once silently to become familiar with the text, once at their natural speaking pace, and once in a line-by-line manner. We recorded the full section, and later divided the audio into chunks that contain individual recordings.

In the second section of the session, we turned on the interactive programming environment and explained its use to the subject. The subject then re-read each of the ten programs, but instead of reading straight through, they were asked to interact with the IDE to try to arrive at the exact printed sample program. They

discovered the phrasing and reading speed that produced the best results, and they used the provided commands to make corrections when the system produced errors. Again, we recorded the full section and later divided up the audio into individual attempts.

In the final section of the session, we provided three programming challenges (e.g., "write a program that reads ten numbers from the user and prints their average."). The subject was supplied with a "cheat sheet" to refresh their memory on the syntax that had been covered in the first two sections, and then was asked to use their remaining time to attempt to write programs to solve the challenges. This section was also recorded for later analysis.

Finally, we asked each subject a few informal questions about their experience and their perception of spoken programming at the end of the session. The responses were not recorded in audio form, but were simply noted.

5.3 A Criticism

In this section, we take a small detour from the description of what we have done to address an important criticism of the design of both our case study and the user study: Real programming is not so linear. An environment that forces the programmer to go line-by-line from top to bottom without deviation is an environment that is not likely to be used for programs larger than a few dozen lines. Unlike our test cases where we first wrote out the intended program and then attempted to dictate it, a real program is typically written in small pieces, and the programmer jumps around between the pieces as inspiration and convenience dictate. Even for the programming challenges where we did not pre-write the intended solution for the subject, the limited editing and navigation capabilities of our spoken programming environment still essentially force the user to attempt to dictate the program start

to finish in a single pass.

We agree that real programs are not generally written in such a linear fashion, and that more advanced editing capabilities are necessary before this environment would be usable for serious programming. Possible future work along these lines is suggested in Chapter 7. However, we do not believe that this lack invalidates the results presented in this dissertation. The concept of using context to improve the recognizer's accuracy is still highly relevant even if chunks of code are dictated or edited out of order.

The analogy of treating program creation as a dialog is less clear-cut if the program is built out of order, but it is not entirely invalid. The language is still constrained by the grammar, and at any particular point in the program, there is still a set of reasonable expected statements that is dictated by the code around that point. For example, it still would not make sense to end a while loop prior to starting it regardless of whether the programmer prefers to write the loop bounds first or fill in the inner statements first.

The thing that changes about the dialog with more advanced navigation is the complexity of the transitions between contexts. Instead of an orderly progression of contexts that mirror the program structure, the user might jump backwards into a previously created context at any time. However, as described in section 4.2.1, we already have a limited form of this situation when the user backs up to make a correction. In order to support restoring the previous context, we already keep track of the complete context that existed at every line of source code. This suggests that it is not too much of a stretch to imagine that a similar implementation might be able to support the necessary jumps into arbitrary prior contexts.

Like the idea of a dialog, scoping and type inference are still relevant regardless of the order of program statement creation. Variables still have a scope and a type even if they're added to the program after the initial dictation. One wrinkle in this

situation is that a programmer might easily add a line to refer to a variable that they plan to go back and define later. Since this is at cross purposes to our proposals here of biasing the recognizer against that situation, the idea initially sounds problematic. We believe it is likely to be solvable with some additional syntax or command to the environment that tells it the programmer intended that situation.

To gain some intuition for how this might work, imagine programming with a partner. If you say, "Now set x to 5" and there isn't an x in the program, your partner is going to be confused. They may say something along the lines of "but there isn't an x," or "did you mean this other variable?" or "where did x come from?" If instead you say, "We're going to want an x. I'll tell you where to put it in a minute, but for now set it to 5," then there is no such confusion. They insert the assignment and proceed, confident in your promise that you'll fix the problem later. Similarly, the extra syntax would tell the spoken environment that you meant to refer to a variable that you haven't told it about yet.

Taken together, these considerations suggest that it should be possible to enable non-linear spoken programming without discarding the ideas we have investigated here. We make the initial step in this dissertation of showing that context is useful in the simple case, and we expect that future work will extend this to more complicated situations.

Chapter 6

Results and Discussion

Now that the preceding chapters have described our programming environment, its features, and the user study designed to measure its effectiveness, we finally have everything needed to present the study results. In this chapter, we first present the outcome of the initial case study. The bulk of the chapter is devoted to a description and discussion of the result of the full user study. Final discussion of the implications of the results and future work building on this study are deferred to Chapter 7.

6.1 Data Analysis and Error Classification

Before moving on to the actual results, we first provide a description of how we turned the raw audio data generated by our subjects into measurable and comparable quantities, as well as an explanation of why these particular measures were chosen.

6.1.1 Procedure

To analyze the data, we first divided the recordings into segments containing a single program reading. We then processed each segment using the programming environment with whatever settings were appropriate for that particular test. By processing the audio files directly rather than replaying through a microphone or asking the participants to re-dictate, the analysis completely eliminates variables caused by environmental factors, such as microphone distance, room/background noise, participant speech variations, etc. Furthermore, even though Sphinx contains a probabilistic language model inside, it produces the same output when given the same input as long as no parameters or grammar rules have changed. This means that the audio files represent a set of completely reproducible textual outputs, and we did not need to average multiple runs or do similar probabilistic smoothing.

Each run of the environment with a particular file produces an instance of program text. We hand-classified each instance based on what was being measured in that test.

6.1.2 Word Error Rate

The specific details of each test will be described below, but first a note about what we have not reported on: In many speech recognition tasks, researchers report fairly standardized metrics such as word error rate (WER). We have chosen not to report these because it is not clear that they are meaningful in this type of environment. Word error rate is typically defined as

$$\text{WER} = \frac{S + D + I}{N}$$

where S is the number of substitutions, D is the number of deletions, I is the number of insertions, and N is the number of words in the correct text. There are two major problems with this in our environment.

First, errors are not independent. For example, if a `while` is missed at the beginning of a statement, not only is the `do` after the guard also going to be missed, but the `end while` that should normally come at the end of the block will also be missed. Thus, a single deletion error is turned into at least 4 errors. Other types of insertions or substitutions could potentially prevent successful processing of the entire remainder of the program. These types of errors are not nearly so devastating for interactive use; the user notices the problem and simply corrects it before continuing.

The second reason WER is not appropriate in this environment is that not all errors are equally important, but the formula weights all errors equally. This is also true for normal English prose recognition, but the effect is magnified for a program. A human reader of an English transcription with some errors can often get the gist of the message, but a compiler has no way to "fill in the gaps" or correct for nonsensical changes.

6.1.3 Low-level Error Classification

Rather than reporting WER, we have come up with a number of error classifications and task classifications that seem pertinent to the goal of spoken programming. First, we have classified the errors in each program into the following five types:

1. Missed utterances (unexpected). This occurs when the recognizer misses an entire utterance and there was no reason to believe it should have done so. This produces an incorrect program and indicates an error occurred in the recognizer.

2. Missed utterances (expected). This occurs when the recognizer misses an entire utterance, but it should have done so as a result of an earlier error. For example, if a `while` statement is missed, the `end while` should also be missed. While

this also produces an incorrect program, it indicates that the grammar-based component of the context is working as expected.

3. Wrong construct. This is an unexpected substitution of one construct for another, such as a `print` statement instead of a function call or a single parameter expression instead of multiple parameters.

4. Wrong variable (first). This occurs when the recognizer produces an incorrect variable name on the first instance of that name.

5. Wrong variable (subsequent). This occurs when the recognizer produces an incorrect variable name on a subsequence instance of that name.

6. Wrong value/literal. This occurs when the recognizer substitutes a valid literal for another literal or a variable of the same type (e.g., 10 instead of n).

Of the errors above, some are expected to be correctable by the context-based features and others are not. Unexpected missed utterances are generally not correctable, because they occur when the recognizer simply fails to produce a result for a particular utterance. Expected missed utterances are not correctable because they are part of the expected behavior of the system. Wrong constructs are largely not correctable, because they are produced when the recognizer generates a wholly incorrect sentence from the grammar for a particular utterance. If the errors are limited to substitutions of variables and literals, they would be classified in one of the other categories. Wrong initial uses of variables may be correctable or uncorrectable depending on the previously used symbols. For example, if the system substitutes an n for an m in an assignment and neither name is in scope, there is no reason to believe that the context-based features we are testing would make any difference in that error. If however, one of n or m is already in use in a way that precludes its use in that assignment, we would expect the context-based features to help reduce instances of that error. Subsequent variable uses are often correctable as long as the

incorrectly generated variable name wasn't also a valid choice. For example, if n and m are both in scope at the same level and both integers, the system does not have any additional context to help distinguish between them, but if one has a different type or a different scope, we would expect our context-based changes to make a difference. Finally, the wrong value errors occur when a legal value or variable is substituted for another. Since the erroneous value is legal at the place of its use, there is no reason to believe that the additional context would steer the system away from it.

6.1.4 Task Classification

When dictating a program, what is the user's real goal? We claim that the goal is to create a program that does whatever the user has in mind. As such, the particular types of errors produced are of less interest to the user than the questions of whether they produced a compilable and/or correct program. After analyzing all the user sessions and categorizing the errors as described above, we realized that it simply produces a sea of numbers that provide no real insight into whether the user is achieving their goal. To support a more insightful analysis of the data, we have also classified each program produced based on three more coarse-grained descriptions: program compiles, program follows the expected structure, and program is functionally equivalent to the original. Further details of these descriptions will be provided with the relevant data below. The raw low-level error counts have not been presented here, but the explanation is helpful to understand what kinds of errors can cause the tasks below to fail.

6.1.5 Statistical Significance of Results

In order to make claims about the performance differences between the spoken environment baseline and the additional context-based features, we first need to determine exactly what we are comparing and how we will test it (in the statistical sense). The goal is to choose a comparison method and statistical test that allows us to determine if the differences are statistically significant, while also ensuring that the differences can be attributed to our context-based features rather than some other environmental factor. As described in section 6.1.1, we eliminate environmental factors by re-processing the same audio file in our spoken environment while varying which features are enabled. Thus, our population of subjects is being sampled multiple times, with the sets of samples for comparison being "programs dictated with baseline features," "programs dictated with scoping enabled," and "programs dictated with scoping and typing" (abbreviated as "baseline," "scoped," and "typed" below). This still leaves open the question of what kind of statistical test will accurately measure the performance differences.

Because users vary widely in speech speed, disfluencies, and other characteristics that will affect their baseline ability to use a spoken system, it is not reasonable to assume that the error rates and entry rates will be normally distributed within each population. This means that we cannot compare users to each other. Even within a single user's set of programs, it is probably not reasonable to assume their errors are normally distributed between programs. Thus, we cannot use the common Student's t-test [42] to compare the different sets of samples, because the t-test requires that the two populations be normally distributed.

However, because of the reuse of audio files, a particular user's performance in dictating a set of programs with one set of features enabled can be directly compared to that same user's performance on the same set of programs using a second set of features. For example, suppose that subject n's audio files produce 5 correct

programs out of 10 dictated programs when run with the baseline, and suppose that subject n's audio files produce 7 correct programs out of 10 dictated programs when run with scoping. Regardless of the exact distribution or causes of subject n's errors, because of the way the audio was re-processed, we can say that the difference of 2 programs is attributed to the addition of scoping.

In this situation, a Wilcoxon signed-rank test [43] is appropriate. In the Wilcoxon test, we do not need to assume that the data are normally distributed. We need only assume that each pair of data points are from the same population, represent independent observations, and that they are comparable numerically. To perform the test, each user's performance in the first sample is compared to their performance in the second sample, and we take the sign of the difference. The differences are ranked based on the magnitude of the difference, the ranks are summed separately for positive and negative differences, and the total is looked up in appropriate critical value tables to determine if the result is statistically significant, i.e., produces a p-values less than 0.05. Intuitively, the result is significant if a large enough fraction of the differences have a positive sign (meaning those users showed improvement). Unless otherwise specified, we have used the Wilcoxon signed-rank test to evaluate the raw results below.

6.2 Case Study: Comparison to Dictating C

The first results are from our initial case study. Recall from the previous description that this involved a single user comparing Dragon speech recognition software versus our spoken programming environment to dictate a simple program. The text of the C and equivalent spoken syntax are given in section 5.1.

Here is a transcript of the words necessary to recreate the C program in Dragon. Each bullet represents the words for a single line. During dictation, we paused at

the end of each line to ensure that the line was correctly recognized. Additional explicit pauses needed to let Dragon recognize the text to that point are indicated with *pause*.

- hash sign include less than sign no caps letter s letter t letter d letter i letter o dot no caps letter h no space greater than sign new line
- no caps letter i letter n letter t *pause* main open parenthesis no caps letter i letter n letter t *pause* no caps letter a letter r letter g letter c comma char space bar star star no caps letter a letter r letter g letter v close parenthesis open brace new line
- tab key no caps letter i letter n letter i *pause* letter i equals sign one semicolon new line
- tab key while open parenthesis letter i less than sign equals sign ten close parenthesis open brace new line
- tab key tab key print no space no caps letter f open parenthesis open quotes percent sign no space no caps letter d close quotes comma letter i close parenthesis semicolon new line
- tab key tab key print no space no caps letter f open parenthesis open quotes backslash no caps letter n close quotes close parenthesis semicolon new line
- tab key tab key letter i equals sign letter i plus sign one semicolon new line
- tab key close brace new line
- tab key return zero semicolon new line
- close brace new line

Here is a transcript of the words needed to recreate the equivalent spoken program in Dragon. Note that even though it is mostly English, the line breaks, tabs, and punctuation still have to be spelled out explicitly.

- define function main taking no arguments as new line
- tab key set letter i *pause* to numeral one new line
- tab key while letter i less than sign equals sign ten do new line
- tab key tab key print letter i new line
- tab key tab key new *pause* line new line
- tab key tab key set letter i *pause* to letter i plus sign one new line
- tab key no caps letter e letter n letter d *pause* while new line
- end function new line

At a glance, it is immediately clear that far fewer words overall are needed to coax the desired program out in the second case. This alone should lead one to expect faster program entry and fewer corrections, and this assumption is borne out by the results, as shown in Table 6.1.

	Time to Dictate Program	Corrections Required
C Program	159 ± 27s	8 ± 6
Spoken Program	69 ± 17s	2 ± 2

Table 6.1: Dictating C and Spoken Language with Dragon Dictate

This demonstrates that even using an unspecialized commercial speech recognition package with no built-in knowledge of programming, simply switching to the spoken syntax makes a large improvement. Speaking the program takes less than half as long on average (35% less time even comparing the worst trial of the spoken

syntax to the best trial of the C syntax), and the average number of corrections needed is also reduced.

This large difference in speaking time can be partly attributed to the relatively fewer number of words needed for the spoken syntax. It can also partly be attributed to the "naturalness" of the mental mapping required for the spoken syntax; even after over an hour of practice on the same program, we found that we kept losing our place when dictating a line of C and had to pause to remember the syntax for various punctuation symbols. Qualitatively, the programming experience was also improved with the spoken syntax. Because we were not focused on speaking out all the punctuation, we were able to pay closer attention to what the code actually said.

Due to the large amount of training required on Dragon and the very strong results, we did not re-run this part of the experiment on any of our other subjects during the user study.

6.3 Case Study: Dragon Compared to the Spoken Programming Environment

As a continuation of the experiment above, we dictated the same spoken program into our programming environment. The results are shown in Table 6.2. Note that the results for Dragon are the same as shown in Table 6.1; they are duplicated here for clarity.

	Time to Dictate Program	Corrections Required
Dragon	69 ± 17s	2 ± 2
Spoken IDE	22 ± 6s	1 ± 1

Table 6.2: Dictating Spoken Program with Dragon Dictate and Spoken IDE

As the table shows, moving from spoken syntax alone to spoken syntax combined

with an environment that incorporates programming context resulted in a dramatic reduction of the time needed to enter the program text. In two of the trials, the spoken IDE required 0 corrections; in the other case, the corrections were due to a misstatement by the speaker rather than an error produced by the system.

Qualitatively, the programming experience was again improved due to the increasing simplicity of the mental map between text and speech. When using the spoken IDE, we did not have to remember to pause between certain symbols and did not have to remember which lines required us to say "numeral one" instead of just "one."

6.4 User Study: Spoken Programming with Context-based Features

The case study described above provides a data point that suggests our claims about spoken programming are on the right track, but the real test comes from the experiments that were enabled by our user study. In the following sections, we will describe the data that resulted from each of the three study components (offline/reading tests, interactive tests, programming challenges) and analyze the results.

6.4.1 Offline Tests

In the offline tests, we processed each audio file three times: once with only the grammar-based context enabled (baseline), once with the scoping turned on (scoped), and once with both scoping and typing turned on (typed). Because the type inference depends on having scope information, there is no separate run for typing without scoping.

Block Structure

The first measure is whether the program produced has the same block structure as the correct program. This means that it has the same number of functions taking the same arguments, the same nesting structure of `while` and `if` statements, and all the blocks are terminated correctly. It does not mean that the symbol names are the same (or even consistent), or even that the program is compilable. For example, if the program has an identical block structure but contains erroneous variable references or invalid function calls, it may not compile at all. Table 6.3 shows the results of this measurement. The maximum number of correct programs for each subject in this test is 10.

Subject	Baseline	Scoped	Typed
1	5	5	4
2	6	6	6
3	8	8	7
4	8	8	9
5	9	8	9
6	8	8	8
7	10	10	10
8	7	8	8
9	9	9	9
10	10	10	10
11	8	8	8
12	8	8	8

Table 6.3: Block Structure Successfully Produced in Contexts

The majority of subjects experienced no change in the number of programs with correct block structure as a result of the context-based features. Of those who saw a difference, some were positive and some negative. However, the differences are not statistically significant. Using Wilcoxon's test, we have $W = 71.5$ with a p-value of 0.52 moving from the baseline to scoped context. Adding typed context, we have $W = 76, p = 0.42$. Finally, going from baseline to fully typed, we have

$W = 75, p = 0.44$.

This result is not much of a surprise, since the context-based improvements we have been considering should affect identifier recognition much more than the overall program structure or basic statement constructs. However, since that is the case, one question that might be raised is why any of the users should have experienced a decrease in correctly structured programs. Moreover, why does the decrease only happen when typing is added? This will be discussed in section 6.4.5.

Compilable

The second measure is whether the resulting program is compilable. This not a superset of the previous block structure measure. A program in which some loops or conditionals get lost may still be compilable even though it would not have the same block structure. The combination of compilable and correctly structured will be separately considered in the next section.

Leaving aside potential missing block structure elements, programs that are compilable may still not be correct. A compilable program could contain incorrect operators in arithmetic or Boolean expressions, incorrect (but legitimate) uses of literal values or variables, calls to incorrect (but defined and properly typed) functions, etc.

Compared to the block structure test, we should expect to see a larger effect for compilable programs from the context-based features. Errors such as using undefined variables or calling non-existent functions should be reduced or eliminated by the scoping, and we can expect errors such as adding a string to a number to be similarly reduced by the addition of typing. The actual results are shown in Table 6.4. Again, the maximum number of correct programs for each subject in this test is 10.

Unlike the block structure test, no users experienced a decrease when adding context-based features. Some had an increase in compilable programs when adding

Subject	Baseline	Scoped	Typed
1	3	4	6
2	5	5	6
3	3	6	7
4	6	6	10
5	3	7	7
6	5	7	8
7	8	8	10
8	5	5	7
9	4	6	9
10	4	6	10
11	5	6	8
12	3	6	9

Table 6.4: Compilable Programs Produced in Contexts

scoping, some when adding typing, and some in both cases. Intuitively, this tells us that the context-based features have the expected effect, but formally we can check using Wilcoxon's test again. We find that $W = 117, p = 3.9 \times 10^{-3}$ for adding scoping to the baseline. Adding typing, we get $W = 125, p = 9.4 \times 10^{-4}$. Finally, going from baseline to fully typed, we get $W = 137, p = 8.7 \times 10^{-5}$. These p-values indicate very strongly that the increase in the number of compilable programs is not due to chance. The scoping and typing are individually statistically significant improvements, and the combination even more so.

We also see in these results that the two types of context we have been considering are able to fix different types of errors. As predicted, the scoping feature gets rid of errors where the recognizer produces a reference to an undefined variable or function, but it cannot help distinguish between similar-sounding identifiers that are both in scope. Adding the typing removes these ambiguities by ensuring that variables are used in a type-compatible way.

Compilable and Structured Correctly

Since compilable programs may not exhibit the correct block structure and vice versa, it is also relevant to compare the outcomes for programs that are *both* compilable and correctly structured. This combination represents programs that compile and follow the structure laid out by the programmer, but they may still not be perfect. The type of bugs that may be present are limited to basic typos like having an incorrect comparison operator in a Boolean expression or substituting an incorrect value where a variable was expected. See Table 6.5 for the results. The maximum number of correct programs for each subject in this test is 10.

Subject	Baseline	Scoped	Typed
1	2	3	3
2	5	5	6
3	3	6	7
4	6	6	9
5	3	7	7
6	5	7	8
7	8	8	10
8	5	5	7
9	4	6	8
10	4	6	10
11	5	6	8
12	3	6	8

Table 6.5: Correctly Structured Compilable Programs in Context

Intuitively, we can see that there are improvements for every subject and no regressions, so the context-based features appear to be a clear improvement. Performing our usual Wilcoxon test to verify this formally, we find $W = 115, p = 6.0 \times 10^{-3}$ for adding scoping to the baseline. Adding typing to this, we get $W = 118.5, p = 3.3 \times 10^{-3}$. And finally, going from the baseline to the fully typed setup, we get $W = 128, p = 6.0 \times 10^{-4}$. Like the previous section, we have a statistically significant improvement on each feature separately, as well as in combination.

Even though the improvements are large, we can see from Table 6.5 that most of the subjects still did not manage to get 10 out of 10 programs. Nevertheless, at this level of correctness, it seems plausible to describe the results as a "working program with a bug" instead of "wrong program." The remaining errors are the types of errors that a normal person typing a program at the keyboard might easily make.

Functionally Equivalent

The final measure for our offline tests is whether the output programs are functionally equivalent to the intended programs. What this means is that the produced program compiles, is structurally identical to the intended program, and contains no incorrect variable names or literal values. The only difference allowed from the input program is renaming variables *as long as the change does not alter the program behavior.* For example, if the recognizer replaces a with k everywhere and k wasn't otherwise used, the generated code and program behavior remains the same and the program is considered functionally equivalent. The results for this measure are shown in Table 6.6. The maximum number of correct programs for each subject in this test is 10.

The results here are not quite so clear-cut. Adding scoping to the baseline, we get $W = 92.5, p = 0.12$, so this is not strong enough to claim significance. Adding typing, we get $W = 78, p = 0.37$, even less strongly supported. But going from the baseline to fully typed, we get $W = 96.5, p = 0.08$, which is just shy of the 0.05 bar of statistical significance.

This level of near-perfect output is obviously the most desirable, and it is disappointing to see that our results in this case do not support a claim of statistically significant improvement. However, this is mitigated by two factors. First, as mentioned in the previous section, it is entirely common and even "normal" for a programmer using a keyboard to produce programs that are not perfect even though

Subject	Baseline	Scoped	Typed
1	1	2	1
2	3	2	2
3	3	4	4
4	5	6	7
5	2	4	4
6	4	5	5
7	8	7	9
8	5	5	5
9	3	3	3
10	3	5	5
11	3	3	4
12	2	4	5

Table 6.6: Functionally Equivalent Programs Produced in Contexts

they compile (e.g., how many security bugs have resulted from an accidental "<" where "<=" was meant?). If this were not true, there would not be areas of research and entire companies dedicated to helping reduce errors and debug programs. Since error-free-on-the-first-try programs are not generally achievable with traditional programming, it is too high of a bar to imagine that spoken programming will produce them.

Second, many of the errors these programs contained turned out to be easily noticed and quickly fixed by the user in interactive use. Due to the nature of the dictation process for this offline part of the study, no corrections could be made, so the number of errors overestimates the number produced in real use of the system. This will be discussed further in section 6.4.2.

Comparison by Programs

In addition to the breakdowns by subject described in the previous sections, it is also useful to see if the performance of the system is affected by the program itself. In order to see if the improvements from the context-based features varied by the

structure of the program as well as by the user, we performed similar analysis broken down by program. Because the combination of correctly structured and compilable programs represents the most useful case, we only performed this analysis for this particular case. The resulting data for the combination of correctly structured and compilable is shown in Table 6.7. In this table, the maximum possible is 12 instead of 10, because we are counting the number of subjects who get each program correct instead of the number of programs each subject gets.

Program	Baseline	Scoped	Typed
1	10	10	10
2	12	12	12
3	10	12	12
4	4	10	10
5	7	11	11
6	3	10	10
7	2	3	10
8	0	0	5
9	4	2	7
10	1	1	4

Table 6.7: Users Completing Correctly Structured Compilable Programs

Calculating our Wilcoxon test, we get $W = 60, p = 0.23$ for the change from baseline to scoped. Adding typing from scoped, we get $W = 59.5, p = 0.24$. Finally, going from baseline to fully typed, get get $W = 77.5, p = 0.019$. Thus, the changes are individually not statistically significant, but the full set of both scoping and typing makes a large enough improvement to be significant.

Looking at the programs individually, we can try to come up with explanations for which features are problematic and which features are helped by the different types of context. The full text for all ten programs can be found in Appendix B, but we will briefly discuss them individually here. Programs 1–3 are trivial programs with no nesting or looping. They simply set a variable of different types and then print it. As the results show, these programs didn't benefit much from the additional context.

Most people got them even in the baseline, and there wasn't much improvement from adding the context (nor much room for improvement).

Program 4 first introduces a `while` loop. This appears to have been a large problem, as the number of people who got it in the baseline drops by 60%. Adding the scoping restored the performance back because the missed variables in the condition are now corrected by knowing what is in scope. Program 5 introduces the `if-then-else` construct and shows a similar story. In program 6, we have both a while loop and an array variable, and the overall result is similar. These three programs still have just a couple of variables of a single type each, but they have started to introduce some structure. Thus, the scoping brings a large improvement, but the additional information from typing doesn't add much more.

Finally, programs 7–10 are longer and more complex. They contain multiple functions with lots of variables and a more deeply nested block structure. Merely knowing the block structure of these programs doesn't help to get all the variables right, and this is borne out by the fact that adding scoping alone didn't make much improvement. Once typing was added, all the mistakes of attempting to add an array variable to an integer or to set an element of an integer variable disappear, and the number of viable programs jumps accordingly. We observed that the `while` loop in program 4 seemed to cause some severe difficulty, and this is empirically confirmed by program 8. Program 8 has a doubly-nested set of `while` loops with an `if-then` block inside that. This was complicated enough that nobody managed to get all the right variable references in place without the help of the typing.

Summary

These offline tests provide a direct comparison of whether the context-based features improve the outcome for a particular user. We found that these features are likely not needed for capturing the basic block structure of the program; most participants

exhibited good performance in the baseline and the differences from the context-based features were not significant.

While the ultimate goal might be to dictate a perfect program, we unfortunately found that the improvements here were also not statistically significant. However, given the fact that people who enter programs via keyboard and mice don't automatically produce perfect programs without tweaking, either, we suggest that this might not be a realistic goal. A more realistic goal is to capture the structure of the user's program and produce compilable output. Subsequent problems are bugs that can be found through normal debugging and testing. Considered on this metric, we found very strong support for the claim that the context-based features help produce programs.

When considering the results by program instead of by user, we found that programs require a certain initial complexity before the additional context has any room to help. Once a program has a small amount of block structure with a couple of variables, the scoping makes a large difference. Once the program passes some threshold in terms of complexity, scoping alone is no longer adequate, either. For these more complex programs, the type-based context makes a substantial difference. The combination of scoping and typing produced significant improvements overall with a p-value of 0.019.

6.4.2 Interactive Tests

The second section of the user study was the interactive tests. As described in Section 5.2, the user was directly interacting with the running programming environment for this part of their session. They were able—and encouraged—to make corrections when the system produced incorrect output. This means that unlike the offline recordings described in the previous section, the interactive sessions can't be replayed through the system with different settings later. The user's interaction is

dependent on the specific behavior of the system at the time they used it.

Instead of replaying each user's recording multiple times to evaluate the context-based features, we randomly divided the participants into two groups. Of our twelve participants, six used the baseline system (group 1) and six used the full system with scoping and typing (group 2). Because of the small size of our study, we did not have any participants record with scoping only. Due to a recording error, interactive data for two of the participants was unusable; thus, we have five people in each group. Within each group, we have numbered the subjects from 1–5 without regard to their numbering in the offline section; thus, subject 1 in group 1 does not necessarily correspond to subject 1 from the offline tests. More importantly, subject 1 in group 1 is not the same person as subject 1 in group 2. The tables below have been divided by a vertical space to help visually reinforce this warning.

Because each participant was encouraged to make corrections as needed, we did not record whether the programs were perfect. For each subject, we recorded only whether they were able to complete each program within a 5-minute time limit. For completed programs, we also recorded how long it took and how many corrections were necessary.

Since we are comparing separate groups of people instead of comparing each participant to themselves, we cannot use the Wilcoxon signed rank test (the Wilcoxon test requires that the data pairs come from the same population). As described previously, we still cannot assume that the number of programs completed or numbers of corrections are normally distributed. We can assume only that the study participants are independent observations and that the numbers are comparable to each other. In this scenario, the Mann-Whitney U test is more appropriate for determining if the performance difference between the two groups is significant, so this will be the statistical test used throughout the following sections.

Programs Completed

The first measure was the number of programs completed. None of the participants in the baseline group were able to complete more than 4 out of 10 programs. All of the participants in group 2 completed at least 8 out of 10 programs within 5 minutes, and 4 out of the 5 completed all 10 successfully. From this description alone, it is clear that group 2 was more successful than group 1. However, to present the result formally, we computed the Mann-Whitney test and found that $U = 25, p = 4.6 \times 10^{-3}$. The data is shown in Table 6.8.

Baseline Group	Completed
1	2
2	4
3	3
4	2
5	4

Scoped and Typed Group	Completed
1	8
2	10
3	10
4	10
5	10

Table 6.8: Programs Completed in Interactive Session

Corrections Needed

In addition to the raw number of programs completed, it is interesting to ask whether the number of corrections necessary to complete the program changed with the addition of the context-based features. The programs were different lengths, so we cannot simply compare the number of corrections directly. Instead, we have normalized by computing a "corrections per line of code" correction rate. In Table 6.9, we present the average correction rate for participants over all programs that each participant completed.

We did not include programs that were not completed in the calculation because it is impossible to know how many corrections might have been needed to finish whatever part remained. We observed that for incomplete programs, subjects had

typically accumulated dozens of corrections before they either gave up or ran out of time. Had all of these corrections been included, the correction rates shown here for the baseline subjects would have been at least an order of magnitude larger.

Baseline Group	Rate (err/line)
1	0.17
2	0.73
3	0.44
4	2.31
5	0.26

Scoped and Typed Group	Rate (err/line)
1	0.25
2	0.32
3	0.21
4	0.20
5	0.14

Table 6.9: Average Correction Rate Interactive Session

Performing the Mann-Whitney test, we find that $U = 5, p = 0.07$. Thus, this result was not statistically significant. We can see from Table 6.9 that the range of correction rates was significantly reduced by the addition of the context-based features (0.18 versus 2.14), but we cannot say that the overall distribution is definitely improved.

However, averaging in this way causes different numbers of programs to be included for each participant. To determine if the rates were affected by the inclusion of the extra programs for group 2, we performed the same calculation on a program-by-program basis for programs 1–3 (since these were completed by most participants). The data is shown in Tables 6.10, 6.11, and 6.12. In these tables, the value "N/A" indicates that that particular participant did not complete that program, while a value of 0.0 indicates that the program was completed with no corrections required.

Baseline Group	Rate (err/line)
1	0.20
2	0.80
3	0.0
4	4.2
5	1.0

Scoped and Typed Group	Rate (err/line)
1	0.60
2	0.40
3	0.0
4	0.0
5	0.20

Table 6.10: Interactive Correction Rate Program 1

Baseline Group	Rate (err/line)
1	N/A
2	1.0
3	0.0
4	N/A
5	0.0

Scoped and Typed Group	Rate (err/line)
1	0.0
2	0.83
3	0.17
4	0.0
5	0.17

Table 6.11: Interactive Correction Rate Program 2

Baseline Group	Rate (err/line)
1	N/A
2	0.0
3	1.33
4	N/A
5	0.0

Scoped and Typed Group	Rate (err/line)
1	0.0
2	0.0
3	0.17
4	0.33
5	0.0

Table 6.12: Interactive Correction Rate Program 3

For program 1, we find $U = 6.5, p = 0.12$. For program 2, we find $U = 8, p = 0.63$. For program 3, we find $U = 7, p = 0.5$. Thus, even comparing directly equal program to equal program, we see about the same results as comparing the overall average correction rates. The span of error rates is smaller with the context-based features enabled, but the overall distribution is not definitively improved.

One notable outlier that jumps out of the tables above is the particularly poor performance of subject 4 from group 1. This person had an error correction rate of more than 3 times the next highest person! Was there something about subject 4 that made their performance so much worse? As it turns out, subject 4 was one of our two female subjects. This will be discussed in more detail in section 6.4.5.

Completion Time

The final comparison we performed between the groups was to see if the context-based features reduce the time needed to dictate a program. Like the corrections, we

cannot directly compare the times. Not only are the programs different lengths, but different users have different natural speech rates, different pause lengths, etc. We normalized the times by dividing the seconds needed to complete the program by the number of lines to yield a seconds per line dictation rate. We averaged the dictation rates across the completed programs, and the results are shown in Table 6.13.

Baseline Group	Rate (sec/line)
1	4.8
2	7.14
3	3.86
4	30.0
5	6.33

Scoped and Typed Group	Rate (sec/line)
1	12.20
2	9.96
3	8.77
4	8.54
5	4.87

Table 6.13: Average Dictation Rates Interactive Session

Calculating our Mann-Whitney test, we find that $U = 18, p = 0.16$. Thus, we see the same picture as Section 6.4.2: The range of dictation rates is tightened up considerably with the context-based features in group 2, but the overall distribution is not improved in a statistically significant way.

As in the previous section, we repeated the analysis of programs 1–3 for the dictation rate measurement to determine if the individual results differed from the average. The results are shown in Tables 6.14, 6.15, and 6.16. A value of "N/A" means the program was not completed.

Baseline Group	Rate (sec/line)
1	4.8
2	5.6
3	2.4
4	44.4
5	13.6

Scoped and Typed Group	Rate (sec/line)
1	28.4
2	7.8
3	3.2
4	3.2
5	3.8

Table 6.14: Interactive Dictation Rate Program 1

For program 1, we get $U = 10, p = 0.34$. For program 2, $U = 10, p = 0.80$. For program 3, $U = 7, p = 0.50$. Thus, we see again that the range of values is smaller,

Baseline Group	Rate (sec/line)
1	N/A
2	12.0
3	2.67
4	N/A
5	3.33

Scoped and Typed Group	Rate (sec/line)
1	25
2	9.67
3	5.17
4	3.17
5	3.67

Table 6.15: Interactive Dictation Rate Program 2

Baseline Group	Rate (sec/line)
1	N/A
2	2.33
3	6.5
4	N/A
5	3.5

Scoped and Typed Group	Rate (sec/line)
1	2.67
2	3.0
3	4.83
4	4.83
5	2.67

Table 6.16: Interactive Dictation Rate Program 3

but the overall change in distribution is not statistically significant.

We can see two oddities in these results. Once again, we see that subject 4 in group 1 shows much worse performance than the others. This is due to the fact that she had to make so many corrections (as we saw in the previous section), so it simply took far longer to get each line correct.

The second big outlier is subject 1 from group 2. This subject did not show the same increased error rate as subject 4 in the previous section, so why did they take so much longer to dictate each line? The fact that subject 1 was tied for the fastest rate in program 3 suggests that this was not merely a matter of speaking more slowly. The actual explanation is that subject 1 had to repeat each line multiple times before it was recognized. Instead of producing erroneous lines that required corrections (and thereby increasing the error counts), Sphinx had so much trouble with most of the utterances that it simply produced nothing. This phenomenon occurred occasionally for every subject, but it was particularly prevalent for subject 1. Perhaps not coincidentally, subject 1 from group 2 was our other female subject.

We will discuss this further in section 6.4.5.

A final observation about these results is the correlation (or lack thereof) between the error rate and the dictation rate. For group 2, other than subject 1, there is a perfect rank correlation between the two, i.e., as the number of errors per line increases, so does the time needed to dictate each line. This makes intuitive sense, since the more time the user spends making corrections, the longer it takes to dictate a program.

Group 1, on the other hand, presents a different story. Aside from subject 4, there is no obvious relationship between the error rate and the time needed to dictate a line. While we do not have a rigorous explanation for this behavior, we believe it may be related to the "strangeness" of the errors produced. With the context-based features enabled, the errors produced by the voice recognizer were typically simple, uninteresting errors such as incorrect variable names or Boolean operators. The user was able to quickly see that a line was incorrect, repeat the line as needed, and move on. With the baseline, however, the errors ranged from simple to completely wacky. For example, one run turned this input:

```
set i to the result of calling m with f and n − 1
set element n of f to z + i
```

into this:

```
set i to the result of calling m with f and 10 − 1
    and 11 and 10 and f and z + i
```

Without the scoping and type information, there is no way to know that m is a function taking two arguments instead of a function taking 6 arguments, so the output produced was a potentially legal statement.

When a particularly weird error came out, some users stopped to puzzle over it and try to figure out how the recognizer could have made that error. Since the

level of strangeness was fairly random, the time taken to study the errors did not correspond to the exact number of errors; thus, the time taken does not have an obvious relation to the number of errors corrected.

Summary

For interactive use, the context-based features make a large difference in helping users complete programs. All of the users who used the context-based features completed more programs than any of the users who did not use the context-based features.

When comparing programs that users in both groups 1 and 2 completed, the variance in rates of corrections per line and seconds to dictate each line were much smaller. However, when compared as an overall distribution, the results were not statistically significant.

The combination of these two observations leads us to speculate that for the programs that were simple enough to be completed with the baseline features, they were also simple enough to not benefit greatly from the context-based features. These same programs are also those that did not show significant improvement in the offline comparisons in Section 6.4.1, so this is not a large surprise. For the programs that were large enough and complicated enough to benefit from the context-based features, the use of context made a very large difference.

6.4.3 Challenges

The third section of the user study was the programming challenges. In this section, we offered the user three small programming challenges, and they attempted to write programs to solve one or more of these challenges. Each of the challenges could be solved with less than 20 lines of code.

We gave the users freedom to solve the challenges in any way they liked, and every user had a different approach. Some users thought things through before they started dictating, while others started dictating immediately and then paused to think after every couple of lines. Some users had many more lines than others. Some users wrote almost exactly the solution we had expected, while others came up with something entirely different.

When counting a successful solution, we simply accepted whatever program was produced when the user stopped and told us they thought it was solved. If they stopped short, we counted that as a failure. Some of the programs actually were solutions to the problems given, but others contained logic errors or compilation problems that went unnoticed by the programmer. This seems no different than what one would expect if asking people to type in a solution with a keyboard instead of dictating it.

We observed that the problems were split between difficulties with using the new language correctly and errors caused by the spoken programming. For the users who seemed more comfortable with the language after the first two sections, they had little trouble dictating programs to solve the challenges. Their errors were largely limited to incorrect variable substitutions, literal values, etc., similar to the errors observed in the offline testing. Those users who seemed less comfortable with the language syntax had significantly more trouble solving the challenges, but it wasn't necessarily because the spoken environment was making errors. In many cases, we observed that they were failing to dictate certain constructs because they were speaking incorrect syntax (e.g., "set function" instead of "define function" or "x equals 5" instead of "set x to 5"). Based on our observation, these users likely would have done much better with more practice. Since we did not control for training effects or attempt to ensure that users fully understood the spoken syntax prior to giving them the challenge programs, we did not investigate this possibility further.

One user did not attempt any challenges due to running out of time during the

first two sections of the session. The other users all attempted at least one challenge. Six users attempted two problems, and one subject attempted all three.

One user did not solve any challenge problems in spite of attempting one, while the others who tried one or more solved at least one. Three of the six users who attempted two problems solved both, and the subject who attempted all three solved all three.

We think this is likely to be another manifestation of the situation described above, i.e., the people who felt confident in the language syntax had less trouble with the first challenge, and the positive feedback from completing it successfully made them more willing to attempt a second or even a third challenge. Conversely, the people who struggled to complete the first challenge program were unmotivated to try any additional spoken programming.

6.4.4 Informal Survey

Before each participant left, we informally asked them a few questions about their experience. Responses were generally positive about the idea of spoken programming, especially for the idea of using it on a tablet or smartphone. Users suggested various potential applications for spoken programming, ranging from RSI relief to rapid prototyping on their tablet to some kind of Google Glass integration.

On the negative side, two users commented that they didn't like the idea of spoken programming at all and wouldn't want to talk to their computer under any circumstances. These particular users were very comfortable with their existing keyboard-based setups and had no interest in considering alternatives. A third person liked the concept of spoken programming, but worried that it might be less efficient than keyboard programming.

Users were also generally positive about the syntax. They found it relatively

intuitive, although the syntax difficulties we observed in the challenges may be at odds with this belief. Users generally had suggestions for additional flexibility in the syntax, particularly syntax for various additional higher-level features. One user wanted a way to switch between program syntax and prose within a single document, à la literate programming. One user wished we had chosen a functional programming style instead of imperative, but he also suggested that would have been an impractical decision.

Users were less excited about the specific environment we asked them to use for the study. The primary comments were about the lack of sophisticated editing facilities. Since the environment was intended as a research tool rather than a production programming language and these omissions were intentional, this was not an unexpected shortcoming. A related comment made by several users was that dictating programs as we asked them to do in the study was "too linear." They felt it was important to be able to jump around in their code freely without completing each individual construct first. Our own programming experience accords with this complaint; we believe that the addition of more voice-controlled editing facilities beyond the simple line correction we provided would address both of these areas.

The users who were the least positive about the environment were those who used the baseline version. They commented more on the voice recognition problems and the frustration of getting through a program, whereas the subjects who used the environment with the context-based features enabled seemed to feel that the voice recognition aspect was adequate. This seems to indicate that once the system reaches some level of accuracy, users don't feel unduly burdened by the need to make an occasional correction.

6.4.5 Other Observations and Unexpected Results

In the process of conducting the study, we observed a few other interesting occurrences that didn't fit into any of the previous analysis. They are presented here as a sort of "laundry list" of extra tidbits.

First, we found that `while-do` statements seemed to be the most problematic statement from a speech recognition standpoint. Every participant had trouble with them. Nobody had any difficulty with the syntax; they were probably one of the most intuitive statements we presented to participants. Nevertheless, `while` statements were rarely produced correctly without at least one correction. Program 8 contains nested while loops, and it was also the program that gave the most problems to participants. The grammar for `if-then` statements is similar to the point of sharing the rules for the condition part of the two statements, but it did not present the same problems. We do not have a definite explanation for this phenomenon, but we did observe that the pronunciation of the word "while" had more variation than most of the other language keywords. The condition was the problem far more often than the word "while" in a `while` statement, but it may still be related.

On an intuition level, we found that our syntax for `set` statements was the most troublesome. The recognizer had little trouble recognizing our `set` *`var`* `to` *`expression`* syntax, but some participants had consistent difficulty remembering it. The expression `x equals 5` was the most common attempted substitution for `set x to 5`, but `define x to 5` and `set function f ...` also showed up surprisingly often. This kind of mental word swapping probably indicates that further practice is needed to become fully fluent in our syntax.

Although we divided errors into "fixable" and "unfixable" in Section 6.1.3, we found that a number of "unfixable" errors were fixed by the addition of the type system. For example, the spoken input "call f with x and n - 2" might have initially been turned into "call f with x / 2". This would have been classified as an unfixable

error, since the "and", "n", and "-" are entirely missing and the "/" has mysteriously appeared. After enabling typing and giving the recognizer the information that f needs two arguments, it suddenly reinterpreted the same sounds into the correct output. We also observed a few instances of entirely wrong constructs being turned into the correct construct after scoping and/or typing was enabled. Not only were identifiers whose rules we explicitly manipulated being recognized differently, but other untouched words in rules were now being processed differently. This phenomenon was initially puzzling, but eventually we decided that what must be happening is that the internal phoneme transitions on the state machine inside Sphinx must be different enough with the context-based grammar rule changes to affect the words around them as well. Unfortunately, this effect was not entirely limited to the positive direction. We also observed three instances of originally correctly recognized words turning into incorrect words after the scoping and typing were added. In particular, this was the explanation for subject 3's decreased performance in Table 6.3.

Last, we found that our two female subjects had a great deal of difficulty using the system interactively. Their results were not noticeably worse than the male participants in the offline tests, but they both had a very difficult time being understood by the system when using it interactively. One female subject consistently produced statements with errors that had to be corrected multiple times before a line was correct. The other had a great deal of difficulty getting Sphinx to produce any output from her speech at all. She would have to repeat a line as many as ten times before it would show up, but then it was often perfect. We do not have any explanation for this behavior. Since their offline results look fine and the same Sphinx recognizer was used in both cases, we cannot even blame a hypothetical lack of female subjects in the Sphinx training data.

Chapter 7

Conclusions and Future Work

This dissertation has presented the design of a new programming language that uses English words and phrases as its syntax. This design reduces the semantic and articulatory distances for users who wish to use speech as an input method for programming, and it also enables reuse of the software techniques and models learned from decades of English prose speech recognition research.

Together with the new language, we have built an Eclipse plugin that provides a programming environment optimized for spoken programming. In addition to the necessary integration of speech recognition software, the plugin incorporates contextual information extracted from the in-progress program as it is being dictated. In this dissertation, we have explored three particular sources of context: the programming language grammar and structure, the visibility of symbols as dictated by the nested block structure and scoping in the program, and type information gained from performing type inference.

Our initial case study demonstrates that state-of-the-art speech recognition software for English prose is not automatically good software for writing programs. Switching from C to an equivalent program written with our spoken syntax, we saw a dramatic improvement in Dragon's performance simply by bringing the user's

speech significantly closer to what Dragon expects to hear. We saw further improvements by repeating the same program in our spoken programming environment where the additional types of context we considered could be brought to bear. This provides intuitive validation of the idea that a specialized environment for spoken programming is important.

On a more formal level, we completed a user study of our programming environment in which we analyzed the performance gains contributed by our three forms of context. We found strong statistically significant improvements in participants' ability to create correctly structured, compilable programs. Unfortunately, use of our environment did not lead to a statistically significant increase in programs that were dictated error-free. Since programmers using keyboards and mice typically do not produce perfect programs without further debugging and editing, we believe that this outcome does not invalidate the idea of spoken programming.

When analyzed on a per-program level instead of a per-participant level, we found that the amount of improvement associated with the context-based features varied based on the complexity of the program. Simple programs with little nested structure could be successfully dictated with or without the additional context. However, dictation of complex programs with multiple levels of nesting and larger numbers of in-scope variables showed large improvements with the additional context. This is not dissimilar to English prose dictation; simple sentences can be produced with a grammar or simple n-gram model, but dictating full paragraphs and larger documents is significantly improved by larger models that incorporate topic modeling and other additional forms of context.

Since real programming is an interactive task rather than a reading task, we also asked our study participants to perform some interactive programming with our environment. Here we found the most striking results of all: All participants who used the context-based features were able to complete more programs than any participants who did not have the context-based features enabled. This result

was statistically significant with $p = 4.6 \times 10^{-3}$. In terms of reductions in errors and improvements in the speed of programming, we found that the range of values exhibited by participants was much tighter, but the overall improvement on these metrics was not statistically significant.

The final task for participants in our study was to write small programs to solve some simple programming challenges. We saw mixed results here, with some participants doing very well and others struggling. While our population size was too small to come up with any definite reasons for this, we observed that the participants who seemed more comfortable with the syntax after the initial reading were the same participants who had more success with the challenges. This suggests that further training could eliminate this particular difference. The experience with the challenges also suggests that this may not be an ideal language for initial programming learners. Our study criteria required that participants have already completed the equivalent of several semesters of programming classes, yet some of them still had trouble with the syntax after 30–40 minutes of initial reading practice.

Taken all together, this research paints a picture of a possible future for spoken programming. Lacking physical keyboards, tablets are currently unsuited for major programming tasks, but being able to enter programs via speech eliminates one major obstacle to their use for that purpose. Even given the idea of speaking programs, speaking C, Java, or other traditional computer languages is likely to run into similar limitation to those found in previous research. Rather than attempting to turn human speech into existing computer languages, we have demonstrated that significantly better spoken results can be achieved by molding the computer language to human speech patterns.

Of course, before our new spoken language could be a serious alternative to Java or C, plenty of further research is needed. There are many possible avenues for further study in this area, but we will mention just a few here. The first obvious area is to perform a more systematic study of the syntax used. As we observed in

section 6.4.5, some statements are more intuitive than others. The most intuitive statement is not always the statement with the best recognizer performance (e.g., `while`) and the statements with good recognizer performance are not always the easiest to remember (e.g., `set`).

For spoken programming to be successful, the initial hurdle of remembering syntax and achieving successful program entry must be overcome. However, once that is in place, most programmers spend more time editing and debugging than on the initial program entry, so navigation and editing are vital areas for future study. One of the most common comments made by our study participants was that they wished for more advanced editing capabilities. Several of the papers mentioned in Chapter 2 have already proposed navigation mechanisms for spoken programming. These mechanisms were designed for navigating programs written in traditional computer languages, but many of the ideas could be applied for our spoken syntax with minimal changes.

Spoken navigation is also an area ripe for consideration of multi-modal interfaces. Saying "go up three lines" may be intuitive, but "select this line" with an accompanying screen touch is probably even more so. Once touch is incorporated, many selection or movement tasks can much more easily be described with a gesture than with a wordy description. One can easily imagine the user circling or tapping variables, lines, or whole functions of interest to tell the editor where to make their spoken changes.

Related to editing, spoken debugging is an important area missing from the current prototype. We are not aware of any literature that covers spoken debugging. It is possible that good results could be achieved by something as simple as adding a few spoken commands for "step over," "step into," "examine *variable*" and similar common debugging commands. It also seems likely that just as spoken programming itself benefits from a syntax optimized for human speech, the process of debugging may benefit from a re-thought from the human perspective.

Once we can edit and debug our programs, we need to be able to link them with other libraries and programs. Since we currently produce Java as intermediate code and run on the JVM, it might make sense to be able to call out to existing Java libraries. In this scenario, we suddenly need a spoken syntax for existing Java code after all. Since this is exactly what much of the previous research into spoken programming covered, potentially those techniques could be integrated into our spoken syntax. Alternatively, we could require that foreign function wrappers be created to map the external libraries to some sort of spoken syntax developed by the wrapper creator. The wrapper itself would be a hybrid of the two languages and would have to be developed in the traditional keyboard-and-mouse environment. This type of requirement is typical of the foreign function interfaces in many existing languages. The first approach would more rapidly enable the use of many existing libraries with minimal wrapper creation effort, but the second approach would plausibly result in superior speech-like syntax in the long run.

The final area of future research we will mention here is higher-level language constructs. The language we developed is around the level of C, with a few additions like automatic memory management and type inference. However, most programming languages today supply significantly more built-in functionality. Since the language is Turing complete, it might be argued that all of these features are merely syntactic sugar, but if that made no difference, people would never have developed anything beyond C. The syntactic sugar is not merely pleasant; in many cases it is a core part of a language and a source of programmer productivity. We suggest that spoken syntax for more advanced type systems and object-oriented programming represents an important next step. Once the programmer can use improved types, more advanced looping constructs are sure to become vital. Beyond there, any number of modern high-level constructs could become candidates for being re-imagined into the spoken programming paradigm.

In this dissertation, we have taken a baby step in the direction of a new way to

think about programming environment design: speech first instead of speech as an afterthought. Tablets and similar devices are here to stay; if programmers aren't already asking to program on their tablets, they will be soon. Giving them a bluetooth keyboard and a Java compiler is unlikely to be a satisfactory answer for long, but we hope that more baby steps in the toward spoken programming will eventually lead to a large jump in the right direction.

Appendices

Appendix A

Complete Grammar

A full BNF grammar for the spoken language syntax follows. This is neither the ANTLR grammar used by the compiler nor the JSGF grammars used by the Sphinx recognizer. Instead, it is an equivalent grammar in a form hopefully more suited for human consumption. It contains various ambiguities (such as order of operations in expressions), but these have been left in place to make the rules simpler to understand.

⟨*program*⟩ ::= ⟨*function*⟩+

⟨*function*⟩ ::= ⟨*func-intro*⟩ ⟨*block*⟩ '`end function`'

⟨*func-intro*⟩ ::= '`define function`' ⟨*ident*⟩ '`taking no arguments as`'
| '`define function`' ⟨*ident*⟩ '`taking arguments`' ⟨*arg-list*⟩ '`as`'

⟨*arg-list*⟩ ::= ⟨*ident*⟩
| ⟨*ident*⟩ '`and`' ⟨*arg-list*⟩

⟨*block*⟩ ::= ⟨*statement*⟩
| ⟨*statement*⟩ ⟨*block*⟩

⟨*statement*⟩ ::= ⟨*assignment*⟩
| ⟨*print-stmt*⟩
| ⟨*read-stmt*⟩
| ⟨*call-stmt*⟩
| ⟨*loop*⟩
| ⟨*conditional*⟩
| ⟨*return-stmt*⟩
| ⟨*empty-stmt*⟩

⟨*assignment*⟩ ::= '`set`' ⟨*lvalue*⟩ to ⟨*expr*⟩

⟨*print-stmt*⟩ ::= '`print`' ⟨*expr*⟩
| '`new line`'

⟨*read-stmt*⟩ ::= '`read`' ⟨*lvalue*⟩

⟨*call-stmt*⟩ ::= '`call`' ⟨*ident*⟩
| '`call`' ⟨*ident*⟩ '`with`' ⟨*arg-list*⟩

⟨*loop*⟩ ::= '`while`' ⟨*bool-expr*⟩ '`do`' ⟨*block*⟩ '`end while`'

⟨*conditional*⟩ ::= '`if`' ⟨*bool-expr*⟩ '`then`' ⟨*block*⟩ '`else`' ⟨*block*⟩ '`end if`'

⟨*return-stmt*⟩ ::= '`return`' ⟨*expr*⟩

⟨*empty-stmt*⟩ ::= '`nothing`'

⟨*bool-expr*⟩ ::= '`not`' ⟨*bool-expr*⟩
| ⟨*bool-expr*⟩ '`and`' ⟨*bool-expr*⟩
| ⟨*bool-expr*⟩ '`or`' ⟨*bool-expr*⟩
| ⟨*expr*⟩ ⟨*bool-op*⟩ ⟨*expr*⟩

⟨*bool-op*⟩ ::= '`=`' | '`<`' | '`>`' | '`<=`' | '`>=`'

⟨*expr*⟩ ::= ⟨*expr*⟩ ⟨*int-op*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*string-op*⟩ ⟨*expr*⟩
| ⟨*function-call*⟩
| ⟨*atom*⟩

⟨*int-op*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`'

⟨*string-op*⟩ ::= '`followed by`'

⟨*function-call*⟩ ::= '`the result of calling`' ⟨*ident*⟩
| '`the result of calling`' ⟨*ident*⟩ '`with`' ⟨*arg-list*⟩

⟨*atom*⟩ ::= ⟨*ident*⟩
| ⟨*array-ref*⟩
| ⟨*string-literal*⟩
| ⟨*int-literal*⟩

⟨*int-literal*⟩ ::= ('`0`'..'`9`')+

⟨*string-literal*⟩ ::= '`the string`' .* '`\n`'
| '`space`'

⟨*lvalue*⟩ ::= ⟨*ident*⟩
| ⟨*array-ref*⟩

⟨*array-ref*⟩ ::= '`element`' ⟨*expr*⟩ '`of`' ⟨*lvalue*⟩

⟨*ident*⟩ ::= ('`a`'..'`z`' | '`A`'..'`Z`') ('`a`'..'`z`' | '`A`'..'`Z`' | '`0`'..'9' | '`_`')*

Appendix B

Programs Used in User Study

These are the ten programs presented to users in the study. Note that several of the programs contain logic errors that would cause problems at runtime, but they can all be entered and compiled.

B.1 Program 1: Intro

```
define function main taking no arguments as
    set X to 3
    print X
    new line
end function
```

B.2 Program 2: Arithmetic Expressions

```
define function main taking no arguments as
    read X
    set Y to 2 * X
    print Y
    new line
end function
```

B.3 Program 3: Strings

```
define function main taking no arguments as
    set X to the string
        hello
    print X
    new line
end function
```

B.4 Program 4: Loops

```
define function main taking no arguments as
    read N
    set I to 1
    while I <= N do
        print I
        new line
    end while
end function
```

B.5 Program 5: Conditionals and Function Calls

```
define function f taking arguments n as
    if n < 2 then
        print 1
        new line
        return 1
    else
        set R to the result of calling f with n − 1
        print R ∗ n
        new line
        return R ∗ n
    end if
end function

define function main taking no arguments as
    read X
    call f with X
end function
```

B.6 Program 6: Arrays

```
define function main taking no arguments as
    read N
    set I to 1
    while I <= N do
        set element I of A to N
    end while
end function
```

B.7 Program 7: Fibonacci

```
define function m taking arguments f and x as
    return element x of f
end function

define function main taking no arguments as
    read X
    set element 1 of f to 1
    set element 2 of f to 1
    set N to 3
    while N <= X do
        set z to the result of calling m with f and N − 2
        set i to the result of calling m with f and N − 1
        set element N of f to z + i
        set N to N + 1
    end while
    print f
    new line
end function
```

B.8 Program 8: Bubble Sort

```
define function sort taking arguments a and n as
    set i to 1
    while i < n do
        set j to i + 1
        while j <= n do
            if element i of a > element j of a then
                set short to element i of a
                set element i of a to element j of a
                set element j of a to short
            else
                nothing
            end if
            set j to j + 1
        end while
        set i to i + 1
    end while
    return a
end function

define function main taking no arguments as
    set i to 1
    set element 1 of X to 0
    while i <= 10 do
        read element i of X
        set i to i + 1
    end while
    set short to the result of calling sort with X and 10
```

```
    print short
end function
```

B.9 Program 9: Miscellaneous Functions

```
define function f taking arguments a and n as
    set x to n + 5
    print x
    return a * 2
end function

define function g taking no arguments as
    print 5
    return 1
end function

define function m taking arguments z as
    set n to 2
    set x to the result of calling f with n and z
    print x
end function

define function y taking no arguments as
    call m with 10
end function

define function z taking no arguments as
    return 2
end function

define function main taking no arguments as
    set i to 1
```

```
    call y
    set c to the result of calling z
    call m with c
    set x to 3
    print the result of calling f with x and i
end function
```

B.10 Program 10: Miscellaneous Functions

```
define function y taking arguments x and n as
    return x * n
end function

define function g taking no arguments as
    print the string
        g
end function

define function c taking arguments x as
    print x
end function

define function main taking no arguments as
    set z to 5
    set d to the string
        hello
    read n
    set i to the result of calling y with z and n
    if i < 20 then
        call g
    else
        call c with i
    end if
    read x
    set element i of m to x + n
    set k to i
```

```
    call c with element k of m
end function
```

Appendix C

Turing Machine Simulator

In this section, we present a Turing machine simulator written in our spoken language. This provides both a fun proof of Turing completeness as well as an example of a larger program. In the following explanation, we follow Hopcroft and Ullman's convention [24] in defining our machine as the 7-tuple

$$M = \langle Q, \Gamma, b, \Sigma, q_0, F, \delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, N, R\} \rangle$$

with the following restrictions:

1. States Q are numbered from 0 to N. Since the number of states is required to be finite, this does not impose any limitation.

2. Tape symbols Γ are numbered from 0 to M. Again, this is not a limitation because the number of symbols is required to be finite.

3. The blank symbol b is 0.

4. As a result of the above, valid input symbols Σ range from 1 to M.

5. The only final state F is 0. States 1 through N are used for computation. Although this appears at first glance to be a restriction on the original definition,

the extension from a set of accepting states to a single accepting state is trivial and does not change the computational power.

6. In additional to the original $\{L, R\}$ moves allowed in δ, we also permit N to indicate "no move." This does not increase the machine's computational power.

As input, the program first reads N, M, and the initial state q_0, in that order:

```
define function main taking no arguments as
        read N
        read M
        read Q
```

It then reads δ as a set of 5-tuples:

1. *current state*: from 1 to N
2. *current tape symbol*: from 0 to M
3. *output tape symbol*: from 0 to M
4. *move direction*: -1 for left, 0 for no move, 1 for right
5. *new state*: from 0 to N

The list of tuples is terminated by reading two zeros (*current state* and *current symbol*). The *state-symbol* pair is used to populate the two-dimensional matrix of δ's input, i.e., $Q \setminus F = (1..N, 0..M)$. This matrix is represented in the three one-dimensional arrays *update*, *move*, and *next* via the standard transform

$$index = state \cdot N + symbol$$

This code is shown here:

```
set I to 0
set done to 0
set S to N * N
set S to S + M
set element S of update to 0
set element S of move to 0
set element S of next to 0
while done = 0 do
    read state
    read symbol
    if state = 0 and symbol = 0 then
    set done to 1
    else
            set row to state - 1
                set index to row * N + symbol
            read element index of update
            read element index of move
            read element index of next
                set I to I + 1
        end if
end while
```

The final input is the initial tape contents. This is read as a list of integers terminated by a 0. The tape contents are stored in the array *tape*. All tape cells that are not written to are considered blank, containing a b symbol (0). Because a computer cannot store an infinite tape, we only write to the specified cells at first. Other cells are written as b only when the tape pointer reaches them. This allows us to simulate an infinite tape up to the limits of the machine. Note that in a real Turing machine, the tape extends infinitely in both directions. To simplify the code,

this simulator only includes positive tape cells. If negative cells are desired, the tape lookup could be transformed from arbitrary n to a positive index by the formula

$$index = \begin{cases} 2n+1 & \text{if } n >= 0 \\ -2n & \text{if } n < 0 \end{cases}$$

in constant time without changing the overall simulator. Thus, this omission does not invalidate this Turing simulator as a proof of Turing completeness. The tape reading code is here:

```
set element 1 of tape to 1
set I to 1
set done to 0
while done = 0 do
    read T
    if T = 0 then
        set done to 1
    else
        set element I of tape to T
        set I to I + 1
    end if
end while
```

Now, the program sets up some bookkeeping variables and performs the following loop until it halts (or forever):

1. Look up the current state and tape symbol in the arrays representing δ
2. Write the symbol from *update* to the tape
3. Move the tape pointer in the direction indicated by *move*
4. Change the state to the new state indicated by *state*

5. Print out the current state and tape contents for the user

6. If the new state is 0, halt. Otherwise, continue

```
set max to I − 1
set pointer to 1
set state to 1
set done to 0
set step to 1
while done = 0 do
    if pointer > max then
            set element pointer of tape to 1
            set max to pointer
    else
            nothing
    end if
    set row to state − 1
    set col to element pointer of tape
    set index to row ∗ N + col
    print the string
        Step
    print space
    print step
    new line
    print the string
        State
    print space
    print state
    new line
    call output with tape and max and pointer
```

```
        set element pointer of tape to
            element index of update
        set pointer to pointer + element index of move
        set state to element index of next
        if state = 0 then
                set done to 1
        else
                nothing
        end if
        set step to step + 1
    end while
```

Finally, the program prints out the final tape contents and exits:

```
        print "Program completed"
        new line
        call output with tape and max and pointer
end function
```

We also use a support function *output* to print the tape contents, including "~" to indicate the current tape pointer:

```
define function output
taking arguments tape and length and pointer as
    print the string
        Tape contents
    new line
    set I to 1
    while I < length+1 do
        set X to element I of tape + 0
        print X
```

```
        set I to I + 1
    end while
    new line
    set I to 1
    while I < pointer do
        print space
        set I to I + 1
    end while
    print the string
        ^
    new line
end function
```

Here is a sample program:

```
1
2
1
1 1 2 1 1
1 2 1 1 0
0 0
1 1 2 0
```

This creates a Turing machine with a single state (called 1) and input symbols (1 and 2). The start state is 1. It has two rules corresponding to seeing each input symbol on the tape in its only non-terminal state. The result of the rules is to replace each 1 on the tape with a 2 until a 2 is reached; then the program terminates.

Based on this analysis and the initial tape contents of 1, 1, and 2, we expect the final output to be 2, 2, 2, with the tape pointer pointing to element 4.

Here is the output from running the sample program:

```
Step 1
State 1
Tape contents
112
^
Step 2
State 1
Tape contents
212
 ^
Step 3
State 1
Tape contents
222
  ^
Program completed
Tape contents
2210
   ^
```

Note that a 0 appears on the tape in element 4 when the pointer reaches it. This indicates that the tape element is blank. The abstract tape has 0 in every cell that hasn't been overwritten by the machine's operation, but since this can't be shown, only the cells that the pointer has passed over contain an actual 0.

References

[1] ANTLR Parser Generator. `http://www.antlr3.org/`, Retrieved May 31, 2013.

[2] Stephen C. Arnold, Leo Mark, and John Goldthwaite. Programming by voice, VocalProgramming. In *Proceedings of the fourth international ACM conference on Assistive technologies*, Assets '00, pages 149–155, New York, NY, USA, 2000. ACM.

[3] Attempto project. `http://attempto.ifi.uzh.ch/site/description/`, Retrieved May 31, 2013.

[4] Andrew Begel and Susan L. Graham. Spoken Programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 99–106, 2005.

[5] Andrew Begel and Susan L. Graham. An Assessment of a Speech-Based Programming Environment. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing. VL/HCC 2006.*, pages 116–120, 2006.

[6] Andrew Brian Begel. *Spoken Language Support for Software Development*. PhD thesis, University of California, Berkeley, Berkeley, California, 2005.

[7] Richard A. Bolt. "Put-that-there": Voice and gesture at the graphics interface. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '80, pages 262–270, New York, NY, USA, 1980. ACM.

[8] Barrett R. Bryant. Object-oriented natural language requirements specification. In *23rd Australasian Computer Science Conference, 2000. ACSC 2000.*, pages 24–30, Aug 2000.

[9] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[10] Alain Désilets, David C. Fox, and Stuart Norton. VoiceCode: an innovative speech interface for programming-by-voice. In *CHI'06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, pages 239–242, New York, NY, USA, 2006. ACM.

[11] Alain Désilets, David C. Fox, and Stuart Norton. VoiceCode Demo Movie (CHI2006). `http://voicecode.sourceforge.net/VoiceCode/uploads/VoiceCode_CHI_Demo_Movie.mov`, Retrieved May 31, 2013.

[12] Alain Désilets, David C. Fox, and Stuart Norton. VoiceCode Programming by Voice Toolbox. `http://sourceforge.net/projects/voicecode/`, Retrieved May 31, 2013.

[13] Edsger W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured programming*, chapter 1, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[14] Dragon for Mac – Speech Recognition for the Mac. `http://www.nuance.com/for-individuals/by-product/dragon-for-mac/dragon-dictate/index.htm`, Retrieved April 14, 2013.

[15] Dragon NaturallySpeaking. `http://en.wikipedia.org/wiki/Naturallyspeaking`, Retrieved May 31, 2013.

[16] Eclipse – The Eclipse Foundation open source community website. `http://www.eclipse.org/`, Retrieved May 31, 2013.

[17] Cameron Elliott and Jeff Bilmes. Computer based mathematics using continuous speech recognition. In *Striking a Chord: Vocal Interaction in Assistive Technologies, Games, and More: CHI 2007 workshop on non-verbal acoustic interaction*, San Jose, CA, April 2007.

[18] Richard Fateman. How can we speak math? *Journal of Symbolic Computation*, 25(2), Jan 1998.

[19] International Organization for Standardization. *C – ISO 9899:1999 6.8.4.1(3)*. Geneva, Switzerland, 1999.

[20] Norbert E. Fuchs and Rolf Schwitter. Attempto controlled natural language for requirements specifications. In *Proc. Seventh Intl. Logic Programming Symp. Workshop Logic Programming Environments*, pages 25–32, 1995.

[21] Norbert E. Fuchs and Rolf Schwitter. Attempto Controlled English (ACE). `http://arxiv.org/pdf/cmp-lg/9603003.pdf`, 1996.

[22] Thomas RG Green and Marian Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD.* Citeseer, 1992.

[23] Thomas RG Green, Marian Petre, and R.K.E. Bellamy. Comprehensibility of visual and textual programs : A test of superlativism against the 'match-mismatch' conjecture. In J. Koenemann-Belliveau, Thomas G. Moher, and S. Robertson, editors, *Empirical Studies of Programmers, Fourth Workshop*, volume 121146, Norwood, NJ, 1991. Ablex.

[24] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 7, pages 146–176. Addison-Wesley Publishing Company, 1979.

[25] Edwin L Hutchins, James D Hollan, and Donald A Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1(4):311–338, 1985.

[26] JSpeech Grammar Format. `http://www.w3.org/TR/jsgf/`, Retrieved May 11, 2013.

[27] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*, chapter 9, pages 285–360. Pearson Education Inc., 2009.

[28] Kinect - Xbox.com. `http://www.xbox.com/kinect`, Retrieved May 31, 2013.

[29] Jennifer L. Leopold and Allen L. Ambler. Keyboardless visual programming using voice, handwriting, and gesture. In *1997 IEEE Symposium Visual Languages*, pages 28–35, 1997.

[30] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[31] Christine Masuoka. Java programming using voice input: Adding java support to voicecode. `http://www.cs.umd.edu/Honors/reports/cmasuoka_HonorsProjectReport.pdf`, Fall 2008.

[32] Alan J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, September 1982.

[33] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[34] Benjamin C. Pierce. *Types and Programming Languages*, chapter 22: Type Reconstruction, pages 317–338. MIT Press, Cambridge, MA, USA, 2002.

[35] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. NaturalJava: a natural language interface for programming in Java. In *Proceedings of the 5th international conference on Intelligent user interfaces*, IUI '00, pages 207–211, New York, NY, USA, 2000. ACM.

[36] David E. Price. Using a 'Wizard of Oz' Study to Evaluate a Spoken Language Interface for Programming. Master's thesis, University of Utah, 2007.

[37] David E. Price, Dana Dahlstrom, Ben Newton, and Joseph L. Zachary. Off To See The Wizard: Using A "Wizard Of Oz" Study To Learn How To Design A Spoken Language Interface For Programming. In *In Proceedings of the Frontiers in Education Conference*, pages 2–23, 2002.

[38] David E. Price, Ellen Riloff, and Joseph L. Zachary. A Study to Evaluate a Natural Language Interface for Computer Science Education. In *In Proceedings of the Workshop of Emerging Technologies for Inquiry-Based Learning in Science, 13th International Conference of Artificial Intelligence in Education*, pages 49–60, 2007.

[39] Tavis Rudd. Using Python to Code by Voice. `http://pyvideo.org/video/1735/using-python-to-code-by-voice`, Retrieved April 19, 2013.

[40] S Shaik, R Corvin, R Sudarsan, F Javed, Q Ijaz, S Roychoudhury, J Gray, and B Bryant. SpeechClipse: an Eclipse speech plug-in. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 84–88, 2003.

[41] CMU Sphinx - Speech Recognition Toolkit. `http://cmusphinx.sourceforge.net/`, Retrieved May 31, 2013.

[42] Student's t-test. `http://en.wikipedia.org/wiki/Student%27s_t-test`, Retrieved April 30, 2013.

[43] Wilcoxon signed-rank test. `http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test`, Retrieved April 30, 2013.

[44] Sam Williams. *Free as in freedom: Richard Stallman's crusade for free software*. O'Reilly Media, Sebastopol, CA, US, March 2002.