Developing a Language for Spoken Programming

Benjamin M. Gordon
Department of Computer Science
University of New Mexico

Abstract

The dominant paradigm for programming a computer today is text entry via keyboard and mouse, but there are many common situations where this is not ideal. For example, tablets are challenging the idea that computers should include a keyboard and mouse. 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. Before tablets can can become truly viable as a standalone computing platform, we need a programming environment that supports non-keyboard programming.

I address this through the creation of a new language that is explicitly intended for spoken programming. Rather than attempting to retrofit spoken syntax or other speech aids onto an existing language, I propose to create a new language that has spoken input designed into the syntax from the start. This will enable fluent spoken code dictation in a manner that feels natural to English speakers.

In addition, productive programming requires more than just convenient syntax, so I describe a supporting editor that uses two types of additional context to increase the speech recognizer's accuracy. First, knowledge of identifier scope allows the editor to dynamically modify its speech model increase the probability of in-scope identifiers. Second, the use of type information can be used similarly to constrain the speech model based on which variables can be passed to functions or used in assignments.

1 Introduction

The dominant paradigm for programming a computer today is text entry via keyboard and mouse. Keyboard-based entry has served us well for decades, but it is not ideal in all situations. People may have many reasons to wish for usable alternative input methods, ranging from disabilities or injuries to naturalness of input. For example, a person with a wrist or hand injury may find herself entirely unable to type, but with no impairment to her thinking abilities or desire to program. What a frustrating combination!

Furthermore, with the recent surge in keyboard-less tablet computing, it will not be long before people want to program directly on their tablets. Today's generation of tablets are severely limited in comparison to a desktop system, suitable for viewing many forms of content, but not for creating new content. 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 ideas of a tablet computer.

What is really needed in these and other similar situations is a new input mechanism that permits us to dispose of the keyboard entirely. Humans have been speaking and drawing for far longer than they have been typing, so employing one of these mechanisms seems to make the most sense. In this research, I plan to consider the problem of enabling programming via spoken input.

Successful dictation software already exists for general business English, as well as specialized fields like law and medicine, but no commercial software exists for "speaking programs." Visual and multimedia programming has been an active research area for at least 30 years, but systems for general-purpose speech-based programming are rare. Several researchers have attempted to retrofit spoken interfaces onto existing programming languages and editors [1,2,4], but these attempts have all suffered from the same problem: existing languages were designed for unambiguous parsing and mathematical precision, not ease of human speech.

This research addresses the topic in two specific ways: through the creation of a new spoken programming language, and through the creation of an editing environment for the language.

2 Related Work

The idea of adding speech support to an existing language is not new. In 1997, Leopold and Ambler added voice and pen control to a visual programming language called Formulate [8].

More recently, Désilets, Fox and Norton created VoiceCode [4] at the National Research Center in Canada. Begel and Graham studied how programmers verbalized code [2]. Based on this study, Begeland Graham developed a spoken variant of Java called Spoken Java. 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 [2,3].

Arnold, Mark and Goldthwaite proposed a system called VocalProgramming [1]. Their system was intended to take a context free grammar (CFG) for a programming language and automatically create a "syntax-directed editor," but the system appears to have never been implemented.

Shaik et al. created an Eclipse plugin called SpeechClipse to permit voice control of the Eclipse environment itself [9]. They permitted dictation of "well-known programming language keywords," but primarily concentrated on providing access to the menu and keyboard commands available in Eclipse.

Outside the realm of traditional programming languages, Fateman considered the task of speaking mathematical expressions [7]. He created a system that produced $\TeX$ output from a spoken form of equations.

3 A New Language

The main difficulty exhibited in the previous vocal programming attempts has been that existing programming languages were not intended to be spoken. They tend to have significant amounts of meaningful but difficult-to-speak punctuation. In addition, the syntax is derived from logic and mathematics, not natural languages. Rather than attempting to create spoken syntax for another existing language, I will sidestep this entire issue by deriving the language syntax from the natural language phrases used to describe programming constructs.

This project is meant to demonstrate the viability of this idea rather than to create a full competitor to any existing language. Therefore, it is important to define a minimal usable subset of possible features that can be implemented. While it is possible to reduce every programming construct down to Turing machines or λ-calculus expressions, this hardly leads to a realistically usable language.

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 make it difficult to test the effectiveness of the syntax unless I limit testers to people with existing functional programming experience. Therefore, to maximize accessibility of the language, it will be designed in the imperative paradigm. However, because it is not a full-featured language, I will omit more advanced features like object-oriented programming and metaprogramming (templates, generics, or otherwise).

3.1 Variables and Typing

The core of any imperative language is state manipulation. Therefore, we must have 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. This language will support string, integer, and floating point variables.

In order to perform computations with its state, the language needs to include statements that can combine and manipulate variables. For numeric variables, I will implement basic arithmetic, following the syntax suggestions of Fateman [7] and Elliott [6]. For strings, sensible basic manipulations are concatenation, substring extraction, and simple search (like the *strstr* function in C).

3.2 Structured Programming

Since Dijkstra's famous letter on structured programming [5], 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 looping constructs.

The language will support definition of simple functions that accept parameters with the types described above. Functions will return single values, again of any of the supported variable types. Because pointer types are not present as described above, all function parameters will be passed and returned by value.

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 will include syntax for indefinite loops and omit definite loops.

It is also 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, the language will also include a built-in syntax for conditional statements. In order to avoid the "dangling-else" problem, it will require all *if* blocks to be explicitly terminated. This will induce an unfortunate awkwardness of speech for very short blocks, but I believe this is an acceptable tradeoff for the large reduction in ambiguity it entails.

Both conditionals and indefinite loops require Boolean tests. Thus, simple Boolean tests and logic must be provided. For all variables, this will include tests for equality. For numeric variables, it will also contain numeric comparisons ($<$, $>$, $\leq$, $\geq$). In order to complete Boolean logic, we will also need to be able to combine these with *AND*, *OR*, and *NOT*.

3.3 Input and Output

In order for a program to produce any useful result, it must be able to perform output. The language will include a simple syntax for printing variables and literals of the available types to standard output. It should not be necessary to support advanced output formatting, but syntax for at least newlines needs to be included.

Similarly, a program must be able to accept input if it is to perform any non-hardcoded calculation. Therefore, the language will provide a mechanism reading a value from the user's terminal and storing it into a variable of an appropriate type. Simple input conversions need to be provided so that programs do not have to treat all external input as text, but these should be transparent to the program. For example, if when reading into a variable that has an integer type, input should automatically be converted to an integer if possible.

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 will be omitted, but this will be an important area for future study.

3.4 Language Summary

Summarizing all of the above, the new programming language will be a garbage-collected, statically-typed imperative language supporting the following programming constructs:

1. String literals and variables
2. Integer and floating point literals and variables
3. Variable assignment
4. Simple arithmetic expressions
5. Simple string operations
6. Function definitions
7. Function calls, including recursion
8. Indefinite loops (*while-do*/*repeat-until* equivalent)
9. Conditional statements (*if-then-else* equivalent)
10. Simple I/O (*print* and *read* equivalents)

This minimalist language provides enough features to solve basic programming problems, such as those that students might be tasked with in their first semester or two of computer science classes. It is not intended to be a production-ready language for real software engineering. Once the concept has been proven to be viable, that type of enhancement will be a potential topic for further research.

4 Editor

Productive programming requires more than merely a convenient syntax. An additional important factor will be 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 text editor to write a memo and edit a photo to be a strange one. Similarly, why should we expect that adding a few voice commands to a primarily keyboard-driven editor will produce an excellent voice-driven editor? At minimum, it must be necessary 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. However, this project will make voice programming faster and more accurate through the use of context. The specific types of context that will be considered are variable scoping and typing information.

4.1 Identifier Scope

The first type of context the editor will be aware of is scoping. 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. Due to locality of reference, a programmer is more likely to refer to a nearby symbol than one several nested scopes away from the current line. Thus, using the scope to make more closely-defined symbols appear closer to the top of the list of alternatives often saves time.

In voice 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, given the types of probabilistic language models used in speech recognition systems, the extension of auto-completion to voice input is then obvious. 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 [4,8], but they do not appear to have used the information in the more sophisticated manner proposed here.

4.2 Type System

A second related use of context is the type system. To save the programmer from the burden of declaring the types of all his 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.

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.

4.3 Code Editing

In addition to dictating new code, an editor must provide editing facilities. In a voice-driven editing environment, it only makes sense for these to be voice-driven as well. In a general English editor, the environment must distinguish between speech that is intended to be dictated and speech that is intended to control the editor. Because this editor will be designed to integrate with a specifically designed programming language, it will be largely possible to choose the editing vocabulary to be distinct from the programming language vocabulary. This allows the editor to unambiguously distinguish the user’s intention in most contexts. In a few areas, such as identifier names, the less restricted input may introduce ambiguities. These situations can be handled in same way as proposed in VocalProgramming [1] and VoiceCode [4].

5 Evaluation

This is a limited language that will support a few fundamental features to demonstrate the vocal programming improvements that are possible by treating voice as part of the design. It is expected that further improvements to the language and the editor to bring them to feature parity with existing mature development environments will be topics for future research once my approach has proven its promise. Thus, I will not consider the completeness of this language or suitability for large-scale software engineering as part of my success criteria.

The primary means of evaluating the success of this project will be whether human users are able to produce working code with minimal errors after a short training session. Once the language and editing environment are in a working state, I will conduct a small study of programmers to demonstrate that the editor makes a difference in the accuracy of speech entry.

In order to enable the study, the editor will be designed so that the context-based recognition improvements can be enabled independently from the basic spoken syntax. The design of the study will be as follows: Participants will first be asked to examine and read a few small programs written in the language so that they can become familiar with the syntax. Once they feel comfortable with the syntax, they will be asked to solve several small programming tasks, such as looping over an array or performing a simple multi-step calculation. I will first obtain samples of user input without the context-based features to establish a baseline for spoken accuracy and performance. I will then enable the context-based improvements and ask participants to enter a few additional programs. This should enable a convincing demonstration that the improvements do in fact improve input accuracy and/or speed.

It would be desirable to additionally compare the final results with those of the most similar previous systems, VoiceCode and Spoken Java/SPEED. Unfortunately, only Begel and Graham provided quantitative performance data about SPEED, with word error rates ranging from 20–50% [3]. I will record error statistics while performing the study so that this can become a comparison point for future work. However, due to the limited data available from previous studies, it is not clear that it would be useful to attempt a detailed error rate comparison against SPEED or VoiceCode.

6 Conclusion

The idea of programming a computer through voice input is not a new one, but the rise of tablet computing has made it more relevant than ever. I have proposed the creation of a new programming language and an associated environment in which voice input is part of the original design rather than an afterthought. Upon completion of the editing environment, I expect that these additions will result in a measurable improvement in the speed and accuracy with which code can be produced via speech.

References

[1] 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.

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

[3] Andrew Begel and Susan L Graham. An assessment of a speech-based programming environment. *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 116–120, 2006.

[4] A Désilets, DC Fox, and S Norton. Voicecode: an innovative speech interface for programming-by-voice. *CHI'06 extended abstracts on Human factors in computing systems*, pages 239–242, 2006.

[5] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[6] Cameron Elliott and Jeff A Bilmes. Computer based mathematics using continuous speech recognition. *Striking a C [h] ord: Vocal Interaction in . . .*, 2005.

[7] R Fateman. How can we speak math? *Journal of Symbolic Computation*, Jan 1998.

[8] J.L. Leopold and A.L. Ambler. Keyboardless visual programming using voice, handwriting, and gesture. In *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on*, pages 28 –35, September 1997.

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