Developing a Language for Spoken Programming

Benjamin M. Gordon
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico 87131
bmgordon@cs.unm.edu

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. I address this through the creation of a new language that is explicitly intended for spoken programming. In addition, I describe a supporting editor that improves recognition accuracy by making use of type information and scoping to increase recognizer context.

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 one of the main ideas of tablet computing.

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 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.

Related Work
Several researchers have attempted to retrofit spoken interfaces onto existing programming languages and editors. Spoken Java provides a Java-like spoken syntax and an associated Eclipse plugin that allow Java to be dictated (Begel and Graham 2005). VoiceCode is a more language-agnostic approach; it defines a spoken meta-language and turns that into program code by filling in language-specific templates (Désilets, Fox, and Norton 2006). VocalProgramming was an early proposal for a “syntax-directed editor” (Arnold, Mark, and Goldthwaite 2000). In VocalProgramming, the user would input a grammar for the language and a voice vocabulary (a list of all possible identifiers, function names, etc.). The system was intended to use this to automatically generate an environment that would allow voice input of the designated language, but it appears to have never been built.

NaturalJava is another attempt to generate Java from spoken input, but it was distinguished from Spoken Java by accepting general English as input (Price et al. 2000). The system extracts case frames from the English input and uses these to generate Java source code.

A New Language
The aforementioned spoken programming environments all suffer from the same problem: existing programming languages were designed for unambiguous text parsing and mathematical precision, not ease of human speech. They typically employ syntax based on mathematics and logic, including significant amounts of meaningful but difficult-to-speak punctuation. Even the amount of whitespace carries meaning in some cases. My research addresses this problem through the creation of a new programming language with a designed-in assumption of spoken input. Rather than attempting to create spoken syntax for these features in another existing language, I will sidestep this entire issue by deriving the textual language syntax from the natural language phrases used to describe programming constructs.

The new programming language will be a simple 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 and string operations
5. Function definitions
6. Function calls, including recursion
7. Indefinite loops (while-dolrepeat-until equivalent)
8. Conditional statements (if-then-else equivalent)
9. Simple I/O (print and read equivalents)

I will create English-like syntax for each of the supported programming constructs. The syntax will be primarily word-based so that the spoken syntax of the language will match the textual representation. As a trivial example, instead of an assignment statement looking like

\[ X := 5; \]

an assignment might be

Set \( X \) to \( 5 \)

Punctuation and symbols will be used only where standardized pronunciations exist for them, such as notation used in arithmetic expressions.

The language will be statically typed, but will employ type inference rather than requiring the programmer to annotate variables and functions with type information. This reduces the amount of syntax that needs to be supported and recognized. It also provides additional contextual information to the editor, as will be described below.

This minimalist language would hardly constitute a credible suggestion for use in any serious modern software development project, but it 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 provides many potential topics for further research.

Programming Environment

Speech-convenient syntax alone is not sufficient to enable productive programming. In addition, a programmer needs an environment that supports spoken entry and editing of program text. Therefore, the second component of this research is an integrated development environment (IDE) that will provide optimizations for speech input.

In particular, context is a critical factor in correctly recognizing speech. The editor will take advantage of this by incorporating language context into its recognition algorithms and/or speech models. There are many types of context that might be of interest, but I am focusing on two specific areas: scoping and typing information.

First, the editor will be aware of scoping. Many existing editors use scoping with auto-completion to suggest variable names, functions, or other symbols that are in scope when the user types the first few characters of the name. 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, voice input is typically ambiguous, forcing the speech recognizer to use probabilistic methods to select the output text based on known patterns of English sentences and prior training corpora.

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. 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. I conjecture that this will reduce the number of recognition errors and improve accuracy.

A second use of context is the type system. To save the programmer from the burden of declaring the types of most variables and functions, 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 will 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.

References