Spoken Programs

Andrew Begel, Susan L. Graham
Computer Science Division, EECS
University of California, Berkeley
Berkeley, CA 94720-1776

Abstract

Programmers who suffer from repetitive stress injuries find it difficult to spend long amounts of time typing code. Speech interfaces can help developers reduce their dependence on typing. However, existing programming by voice techniques make it awkward for programmers to enter and edit program text. To design a better alternative, we conducted a study to learn how software developers naturally verbalize programs. We found that spoken programs are different from written programs in ways similar to the differences between spoken and written English; spoken programs contain lexical, syntactic and semantic ambiguities that do not appear in written programs. Using the results from this study, we designed Spoken Java, a semantically identical variant of Java that is easier to say out loud. Using Spoken Java, software developers can speak more naturally by verbalizing their program code as if they were reading it out loud. Spoken Java is analyzed by extending a conventional Java programming language analysis engine written in our Harmonia program analysis framework to support the kinds of ambiguities that arise from speech.

1. Introduction

Many programmers who suffer from repetitive strain injuries (RSI) and other more severe motor impairments have difficulty staying productive in a work environment that all but requires long hours typing code into a computer. We are helping to lower these productivity barriers by enabling developers to use speech to reduce their dependence on typing. To program using speech, the programmer must be able to verbalize both the program and the actions taken in the programming process. In this paper, we address verbalization of the programming language. A spoken command language is a different problem that is not described here.

Many possible verbalizations of written text are amenable to speech recognition analysis: simply spelling out every letter or symbol in the input, speaking each natural language word, describing what the text looks like, or paraphrasing the text's meaning. Spelling every word and symbol or describing the text is tedious and requires prescriptive input methods to which humans would find it difficult to conform [16]. On the other hand, excessive paraphrasing or abstracting the meaning of written content may leave too many details unspecified, and be incomprehensible to a non-expert.

Programming languages exist in a very similar space to natural languages, save two significant differences. Unlike natural languages, which have been spoken since the beginning of time and written for several thousand years, programming languages have only a written form. Consequently, there is no naturally evolved spoken form. Programming languages are also structured differently from natural languages to be much more precise and mathematical. Punctuation, spelling, capitalization, word placement, sometimes even whitespace characters are critical to the proper interpretation of a program by a compiler. Those details of the written form must be inferred from the spoken form.

To design a spoken form of a textual programming language, we need to shed light on the following questions: What would a programming language sound like if it were spoken? How different would it be than the language's written form? If a particular programming language could be spoken, would all programmers speak it the same way? Would programmers who speak different native languages speak the same program in different ways? Programmers who verbalize only a program's natural language words might cause the spoken program to become completely ambiguous. What would be a natural way to speak a programming language that also has a tractable, comprehensible, and predictable mapping to the original language?

Our goal is to enable input that is natural to speak, but at the same time formal enough to leverage existing programming language analyses to discern its meaning. This point in the design space retains some ambiguity, but limits it so that analysis of the language is still feasible. Note that our work is *not* about programming in a natural language using natural language semantics [10, 11, 12], but is about using features of natural language to simplify the verbal input form of a conventionally designed programming language.

To learn how to design our new input form, we conducted a study of programmers to identify how they would speak Java program code without any training or advanced preparation. We found that there are some significant differences between written and spoken code, categorizable into roughly four areas: the lexical, syntactic, semantic, and prosodic properties of input. There is considerable lexical ambiguity, since spoken text does not include spelling, capital letters or an indication of where the spaces in between the words belong. Syntactically, the punctuation that helps a compiler analyze written programs is often unverbalized, leading to structural ambiguities. In addition, some phrases from the Java language prove difficult to speak out loud due to differences in sentence structure with English. Semantically, programmers speak more than the literal code; they paraphrase it, and talk *about* the code they want to write. Finally, we found that prosody is often used by native English speakers to disambiguate similar sounding phrases, but is not employed by non-native speakers.

Based on the study results, we designed a new dialect of Java, called Spoken Java, that more closely matches the verbalization in our study than does the original Java language. In this program verbalization, programmers speak the natural language words of the program, but must also include verbalizations of some punctuation symbols. Spoken Java is not a completely new language – it has a different syntax, but it is semantically identical to Java. In fact, the language grammar is a superset of Java, with only eleven extra grammar rules. Each of these additional rules maps easily onto a Java rule. This syntactic similarity makes it possible for semantic analyses based on parse tree structure to be constructed from analyses built for the original Java language without many changes.

Moving towards this more flexible input form introduces ambiguity into a domain that heretofore has been completely unambiguous. Spoken Java is considerably more lexically and syntactically ambiguous than Java. We have developed new methods for managing and disambiguating ambiguities in a software development context. In our new SPEED (SPEech EDitor) programming environment, lexical ambiguities such as homophones (words that sound alike) are generated and passed to the parser. Given a program with lexical ambiguities and missing punctuation, the parser can construct a collection of possible parses with all possible interpretations of the input [3]. Next, we exploit knowledge of the program being written to disambiguate what the user spoke and deduce the correct interpretation. Using program analysis techniques we have adapted for speech, we use the program context to help choose from among many possible interpretations of a sequence of words uttered by the user. Our research on the semantic aspects of that analysis is still in progress.

In this rest of this paper, we discuss the experiment we conducted, provide examples of what programmers said for different kinds of language constructs, and discuss what these results mean and what they imply about language design for a spoken programming language. We then present our design for Spoken Java and give an overview of the analysis required to understand it. The paper concludes with a discussion of related work and a short summary.

2. How Programmers Speak Code

We designed a study to begin answering the questions raised earlier. We asked ten expert programmers who are graduate students in computer science at Berkeley to read a page of Java code aloud. Five of them knew how to program in Java, five did not. (The latter students knew other syntactically similar programming languages). Five were native English speakers, five were not. Five were educated in programming in the U.S.A., five were educated elsewhere.

The Java code was chosen to contain a mix of language features: a variety of classes, methods, fields, syntactic constructs such as while loops, for loops, if statements, field accesses, multi-dimensional arrays, array accesses, exceptions and exception handling code, import and package statements, and single-line and multi-line comments.

Each study participant was asked to read the code into a tape recorder as if he or she were telling a second-year undergraduate Java programming student what to type into a computer. We chose this instruction over others to try to anticipate the capabilities of the analysis system. We did not want to have the participant assume that the undergraduate knew the content of the code in advance, nor did we want the participant to assume that the listener was completely Java- or computer-illiterate.

The recordings were transcribed with all spoken words, stop words, and fragmented and repeated words. Words with multiple spellings were written with the correct spelling according to the semantics of the program.

For the most part, despite different education backgrounds or lack of knowledge of Java programming, all ten of the programmers verbalized the Java program in essentially the same way. However, each programmer varied his or her speech in particular ways – each had his or her own style. The variations and implications for subsequent analysis are summarized as follows.

2.1. Spoken Words Can Be Hard to Write Down

On a lexical level, most programmers spoke all of the English words in the program. Mathematical symbols were verbalized in English (e.g. $>$ became "is greater than"). There was some variation among the individuals on the words used to say a particular construct. For example, an array dereference `array[i]` could be "array sub i," "array of i," or "i from array." Here "sub", "of" and "from" are all synonyms for "open bracket." A given punctuation could be either "dot" or "period," either "close brace" or "end the for loop."

Several classes of lexical ambiguity were discovered during the transcription process.

- Many of the words spoken by participants are homophones, words that sound alike but have different spellings. In the case of homophones, the same word is recognized by a speech recognizer in several different ways. For instance, "for" could also be "4", "fore" or "four". The language token can be interpreted depending on context (for example, the keyword "for", the number "4" or the identifiers "fore" and "four"). Likewise, $<$ spelled "less than" is a keyword, but as "less then" is a keyword followed by an identifier.

- Capitalization was not verbalized except sometimes as a comment about an identifier, such as "that's class with a capital c". (The analysis must then determine whether the speaker said the letter 'c' or the word 'see'). Most programming languages are case-sensitive – the inability to easily verbalize capitalization causes an ambiguity in which there are two visible identifiers with the same spelling having different capitalizations.

- Spaces between words are implied when the participant is speaking, but when an identifier is made up of several concatenated words, it was unclear whether spaces were intended. For example, "drop stack process" was spoken for `dropStackProcess`. The inability to easily specify where the spaces ought to go between words and the abundance of multi-word identifiers means that any contiguous sequence of words or numbers may constitute a valid identifier.

These ambiguities combine to cause an explosion of possible interpretations of the input stream. Those ambiguities must be resolved prior to compilation. Unlike a human listener who can understand the intent of speech that contains mistakes, a program compiler cannot compile code containing any mistakes – the slightest error, for example, a misplaced character or misspelled name, can render the entire program invalid.

2.2. Written Code Can Be Hard To Say

There were many stop words, false starts, restated expressions and statements, and stream of consciousness utterances sprinkled throughout the spoken code. These speech patterns were particularly common from participants less familiar with Java.

We found that native English speakers had no trouble verbalizing partial words (which were made up of pronounceable syllables) (e.g. `tur` and `pat`) or verbalizing abbreviated words (e.g. `println`). Non-native English speakers often spelled out these partial or abbreviated words.

2.3. One Utterance Represents Many Structures

Much written punctuation was omitted when spoken, for example the dot in a qualified name `object.stack`, the parentheses indicating a method call `e.printStackTrace()`, the comma separating arguments to a method call, or the semicolon at the end of a statement.

Sometimes punctuation was verbalized in context-specific ways. For example, to declare the constructor `Pool(Class kind)`, one person said "constructor pool takes arguments of class kind" (other participants used similar phrasings). "No arguments" was used as a synonym for two matching parentheses with nothing in between, as part of a method declaration or call. "End function," "that finishes the method," "close class," and "end for," were context-specific synonyms for a right curly brace.

Some punctuation was inconsistently verbalized across programmers, and even from the same programmer for different lines of code. For example `System.out.println()` was verbalized on one line as “System dot out dot print line,” and on the next line as “System out print line” (omitting the dots).

Usually, only one element of a pair of matching punctuation symbols was verbalized. For example `array[i]` was expressed as “array sub i.” Here “sub stands in for the left bracket, but the end of the subscript is not verbalized. Ending a while loop was verbalized as “close while,” but no words indicated the open brace at the beginning of the while loop body. Single-line comments were demarcated at the beginning by “begin comment,” but not demarcated at the end (where a carriage return would indicate the end). Multi-line comments, however, were always demarcated at both ends. In many instances, the close brace ending a block would be conflated with the beginning of the next construct; the speaker might say “and then we have a new method,” or “next method.”

Punctuation is used by written programming languages to precisely demarcate program structures. Removing or mangling the punctuation makes the structure of the code ambiguous (e.g. “foo bar” could be `foo.bar`, `foo(bar)`, or `foo().bar()` to name a few possibilities). These ambiguities can combine to make a spoken program difficult to understand.

2.4. Abstraction is Natural

When programmers discuss code with one another, they talk in terms of constructs such as methods, if-statements, or classes and semantic properties such as scope or type, rather than in terms of textual entities. Sometimes they speak program code as it is written, and sometimes they talk *about* code (called meta-coding). The instructions in our study were explicitly chosen to instruct the programmers to speak the program code itself, rather than to describe what it should look like. However, some programmers spoke more than just the literal code; they paraphrased patterns they saw. For instance, they said “All these are just assignment initializations of null. array dot p a t, array dot t u r, array dot o b s...,” or alternatively, “set all the fields of array to null.” Some speech was meta-code: “The first member of the class is...” “And then there’s a forward declaration of the class kind.” After describing a few fields, one programmer stated “these are all members.” When describing the beginning of a pattern of code, a programmer said, “Let’s initialize a bunch of array’s members.”

We see that abstraction is natural: Speakers identify and describe patterns, rather than their instantiations. When humans communicate with one another, they explain concepts at high-level first, and only drop down to a more detailed level if the first explanation is not understood. When programmers paraphrased the code, they abstracted low-level details into a shorter description of how they wanted the code to appear. By supporting this more concise form of input, we would be able to achieve immediate improvements in productivity – for each phrase spoken by a programmer, many lines of code could be written. In addition, before and after a perceived pattern, programmers described what they were about to do, or what they had just done. This kind of speech act indicates the programmer’s immediate intention. It can be exploited by humans to contextualize the utterance and predict its content. A programming system could use these as predictors for code utterances and instantiate code templates for the programmer. Our work does not yet take full advantage of this possibility.

2.5. Prosody Disambiguates

Vocal expression is as important as it is in natural language: Speakers use prosody (volume, timbre, pitch, and pauses) and vernacular to convey meaning. Prosody was used to distinguish between similar-sounding program constructs, for example, “array sub i plus plus” could mean `array[i]++` or `array[i++]`. Note that the left bracket is verbalized, but the right bracket is not.

Native English speakers had different speech patterns than some non-native English speakers. Native English speakers used prosody to indicate a left or right punctuation symbol when it was not otherwise verbalized. They verbalized the first construct in the previous paragraph as “array sub i <*pause*> plus plus” and the second as “array sub <*pause*> i plus plus”. The pause indicates that the terms before the pause are not to be grouped with the terms after the pause. Some non-native English speakers do not have the same familiarity with English prosody. When such a speaker encountered the array dereference ambiguity, he or she completely rephrased the first form as “increment the i_{th} value of the array.” Prosody has limited power in this case – it takes the place of either the left or right punctuation mark in a pair (brackets, parentheses, or braces), but cannot represent two or more punctuation marks (which would be required were there three or more groups of words to be distinguished).

The semantic use of prosody is limited mostly to native English speakers; many non-native English speakers who speak English typically use the prosody of their native language, in which pauses, in particular, do not

```
for(int i = 0; i < 10; i++) {
   x = Math.cos(x);
}
```

(a)

```
for int i equals zero
        i less than ten
        i plus plus
   x gets math dot cosine x
end for loop
```

(b)

Figure 1. Part (a) shows Java code for a for loop. In (b) we show the same for loop using Spoken Java.

```
public class Shopper {
   List inventory;
   public void shop(Thing toBuy) {
      inventory.add(toBuy);
      System.out.println(toBuy.toString());
   }
}
```

(a)

```
public class shopper
    list inventory
    public void shop
      takes argument thing to buy
        inventory dot add to buy
        system out print line
            to buy dot to string
end class
```

(b)

Figure 2. Part (a) shows Java code for a Shopper class with a shop method. In (b) we show the same Shopper class and method using Spoken Java.

hold the same meaning. In our experiment, we interviewed Indian and Chinese graduate students who were non-native speakers, and none of them used the same prosody as the native English speakers. It would be interesting to see whether there are speakers of other languages who are able to employ pausing in a way that could be used for programming.

3. Spoken Java

Our goal in conducting this study was to understand how to design naturally verbalizable alternatives to spoken programming languages. The lessons we learned helped us create Spoken Java, a dialect of Java that has been modified to more closely match what developers say when they speak code out loud. Spoken Java is designed to be semantically equivalent to Java – despite the different input form, the result should be indistinguishable from a conventionally coded Java program.

Several features of Spoken Java were added to address the concerns brought up during the study. Most punctuation is optional, and all punctuation has verbalizable equivalents. Each punctuation mark may have several different verbalizations, both context-insensitive (e.g. "open brace") and context-sensitive (e.g. "end for loop"). We have reversed the phrase structure for the cast operator to better fit with English (e.g. "cast foo to integer") and provided alternate more natural language-like verbalizations for assignment (e.g. "set foo to 6") and incrementing or decrementing a value (e.g. "increment the ith element of a" in place of "a sub i plus plus").

Figure 1 shows an example of how a Java program might be entered in Spoken Java (carriage returns in Spoken Java are written only for clarity). Note the lack of punctuation, the verbalization of operators ("less than" and "equals"), an alternate phrasing for assignment, the verbalization of the "cos" abbreviation, and the assumption of correct spelling for "x" and "i".

Figure 2 illustrates more program structure. Note the lack of capitalization, separation of words "to" and "buy," (and "print" and "line"), the assumed correct spelling for every word (which can not be assumed as the user speaks the code), the expansion of the abbreviation "ln" to "line," the optional punctuation character "dot," and the overall lack of braces and parentheses. Also take notice of the lack of a right parenthesis or suitable synonym after "thing to buy" in the method declaration parameter list.

In these figures, the program fragments appear out of context. When Spoken Java is used in a programming environment, the programmer has visual feedback to indicate how Spoken Java is being translated to Java.

```
for(int i = 0; i < 10; i++) {
   ...
}
```

(a)

```
for loop ... after left paren ...
declare india of type integer ...
assign zero ... after semi ...
recall one ... less than ten ...
after semi ...
recall one ... increment ...
after left brace
```

(b)

Figure 3. To get the for loop in (a), a VoiceCode user speaks the commands found in (b).

4. Spoken Programming Analyses

Our programming by voice system [2] consists of a programming language editor called SPEED (for Speech Editor), and an associated program analysis framework called Harmonia [4], which are both embedded in the Eclipse development environment [6]. A user begins by speaking Spoken Java code into the editor via a commercial speech recognizer. Once the words have been translated to text, they are analyzed by Harmonia. Harmonia's lexical analysis can recognize and handle homophones, miscapitalized words, and arbitrarily concatenated words. These potentially ambiguous lexemes are passed to the parser, which can handle ambiguous structures caused by missing or optional punctuation in the input stream. The many resulting structural interpretations of the input are passed to the semantic analysis engine, which uses the program context to disambiguate them and choose the legal interpretations. Once one or more interpretations have been deduced, they are translated from Spoken Java into Java, and written into the editor. The user can either choose among interpretations or wait until they are resolved by other changes.

Spoken Java is defined by a lexical and syntactic specification language in the XGLR parsing framework [3]. Motivated by the language used by the programmers in the study, the lexical specification supports multiple verbalizations by allowing many regular expressions to match the same token. The grammar is similar to a GLR [20] grammar for Java, but contains eleven additional productions to support three features: a) lack of braces around the class and interface bodies, b) different verbalizations for empty argument lists than for lists of at least one argument, and c) an alternate phrasing for assignment. Each of these additional productions naturally maps to a structure in the Java grammar.

Semantic analysis is written in a variant of C++ and, for the most part, reuses the semantic analysis written for the standard Java programming language. Ambiguities arising from homophones and missing punctuation are resolved in a semantic analysis engine that extends the Visibility Graph [9], a graph-based data structure designed to resolve names, bindings and scopes. The extensions support incremental update (for use in an interactive programming environment) and ambiguity resolution. Two translation modules can translate code back and forth between Java and Spoken Java, provided that the ambiguities are resolved first. If there are multiple interpretations, each is translated separately.

5. Related Work

Commercial speech recognition tools are poorly suited for programming tasks because they are based on statistical models of the English language; when they receive code as input, they turn it into the closest approximation to English that they can. While some disabled programmers have successfully adapted the command grammars that drive speech recognition for programming, the resulting programming tools accept only a prescriptive form of input and provide limited flexibility for ways of programming not anticipated by the tools' authors. Merely speech-enabling text editors, as has been done by IBM, Scansoft and by contributors to public domain software [15, 17] is not enough to support software development tasks. To perform these activities by voice, developers need to speak fragments of program text interspersed with navigation, editing, and transformation commands.

Recent efforts to adapt voice recognition tools for code dictation have seen limited success. Command mode solutions, such as VoiceCode [5, 21], suffer from awkward, over-stylized code entry, and an inability to exploit program structure and semantics. An example using VoiceCode to enter a for loop is shown in Figure 3. The commands are interpreted as follows.

1. **for loop**: Inserts a for loop code template with slots for the initializer, predicate and incrementer.

2. **after left paren/semi/left brace**: Command to move to the next slot in the code template. Analogous commands exist to move to the previous slot. Once all slots have been filled in, future navigation is based on character distance and regular expression searches.

3. **declare india**: Creates a new variable named "i." Most speech recognizers require the speaker to use the military alphabet when spelling words.

4. **of type integer**: A command modifier to "declare", that adds the type signature to a declaration.

5. **assign zero**: Assignment in VoiceCode is "assign," not "equals."

6. **recall one**: Identifiers in VoiceCode can be stored in a cache pad, a table of slots each of which is addressable by a number from one to ten. To reference a previously verbalized identifier, the user says "recall" and the number of the slot.

7. **increment**: VoiceCode's way to say "plus plus."

More recent work has shown that keyword-triggered code template expansion and context-sensitive detection for when the user is saying an identifier can ease some of this awkwardness [19].

Taking a different approach, the NaturalJava system [13, 14] uses a specially developed natural language input component and information extraction techniques to recognize Java constructs and commands. This is a form of meta-coding, where the user describes the program he or she wishes to write instead of saying the code directly. Parts of that work are promising, although at present there are restrictions on the form of the input and the decision tree/case frame mechanism used to determine system actions is somewhat ad hoc. Worse, the tool is not interactive, but rather a batch processor that produces code only after the programmer has described the entire section of code.

Arnold, Mark, and Goldthwaite [1] proposed to build a programming-by-voice system based on simple syntax-directed editing, but their approach is limited and it is no longer being pursued.

An important part of programming is entering mathematical expressions. Fateman has developed techniques for entering complex mathematical expressions by voice [7] that can be used in our speech editor.

Voice synthesis has been appled to speaking programs. Francioni and Smith [8, 18] developed a tool for speaking Java code out loud for blind programmers. Punctuation is verbalized in English, and structure beginnings and ends are explicitly noted (with associated class and method names when applicable). Modulation of speech prosody is used to indicate spacing, comments and special tokens or structures. The design of Spoken Java incorporates several of the design features from this auralization of Java.

6. Future Work

This study looked at programmers speaking prewritten code that they read off a piece of paper. There was no visual or auditory feedback of their progress through the program, nor any way to verify the correctness of the program they spoke. In addition, the program was spoken linearly from top to bottom, which is different from the way most programmers create new code. Some software developers plan the interface to their code before they write the implementation; some write one function and test it before writing the next. Each of these styles would require a speech system to accept partial code or code out of context; supporting the analysis of spoken incomplete or incorrect programs is vital to a usable solution. Our current prototype does not do that. While we feel that we have identified the spoken language used by the study participants for code authoring, our understanding of what kinds of errors they make will require further study.

Many coding situations do not involve simple code dictation by sight, but code composition on the fly. We plan to do another study to look at how programmers speak code spontaneously when asked to write a solution to a coding exercise. We will ask programmers to build a data structure and associated algorithms, and then have them modify the data structure and update the rest of the program. Not only will this show us whether the language used in spontaneous speech is similar to that used to speak the pre-written code, it will also help show the kinds of commands the programmers use to manipulate the code and the editor, as well as illustrate the kinds of ambiguities that result from non-linear code entry.

Repetitive strain injuries from keyboard and mouse use are a motivator for this work, but speech interfaces are not problem-free. Voice strain is a very real problem that heavy users of speech recognition often encounter while they adjust to using a speech environment. Techniques have been developed to help avoid voice strain such as maintaining proper hydration, speaking in a soft voice (while turning up the gain on the microphone), and taking frequent breaks. These techniques are as important to the voice programming training period as learning to use the analysis system.

7. Conclusion

Programming by voice systems can be a viable alternative to keyboard-based programming environments, especially for those suffering from repetitive strain injuries. By first learning how programmers naturally verbalize code and then developing a formal spoken code analysis system based on the lessons we learned, we are taking one of the first human-centric approaches to achieving the goals of this field. Our study has revealed valuable information about the kinds of ambiguities that emerge from spoken programming (which do not appear when using a keyboard), about the use of voice expression and prosody for disambiguation, about the differences between native English and non-native English speakers, and about the human tendency toward abstraction over verbalization of details. Based on these lessons, we have designed a new dialect of Java, called Spoken Java, which is easier to speak out loud. We have used programming language tools to formally describe Spoken Java, and have enhanced these tools to support the kinds of ambiguities that arise from spoken programs. Finally, we are embedding our analyses into SPEED, a speech-based program editor that can use these program analyses to disambiguate what the programmer said and truly enable programming by voice.

Acknowledgments

We thank Michael Toomim for his valuable feedback on drafts of this paper. Our research has been supported in part by NSF Grant CCR-0098314 and by an IBM Eclipse Innovation Grant.

References

[1] S. C. Arnold, L. Mark, and J. Goldthwaite. Programming by Voice, VocalProgramming. In *ASSETS*, pages 149–155. ACM, 2000.

[2] A. Begel. Programming by voice: A domain-specific application of speech recognition. In *AVIOS Speech Technology Symposium – SpeechTek West*, February 2005.

[3] A. Begel and S. L. Graham. Language analysis and tools for ambiguous input streams. In *Fourth Workshop on Language Descriptions, Tools and Applications*, 2004.

[4] M. Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report UCB/CSD-01-1149, Computer Science Division – EECS, University of California, Berkeley, 2001. M.S. Report.

[5] A. Desilets. Voicegrip: A tool for programming by voice. *International Journal of Speech Technology*, 4(2):103–116, June 2001.

[6] Eclipse. http://www.eclipse.org.

[7] R. Fateman. How can we speak math? http://www.cs.berkeley.edu/ fateman/papers/speakmath.pdf, June 2004.

[8] J. Francioni and A. Smith. Computer science accessibility for students with visual disabilities. In J. Impagliazzo, editor, *Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education (SIGCSE-02)*, volume 34, 1 of *SIGCSE Bulletin*, pages 91–95, New York, Feb. 27– Mar. 3 2002. ACM Press.

[9] P. Garrison. *Modeling and implementation of visibility in programming languages*. PhD thesis, University of California, Berkeley, 1987.

[10] H. Liu and H. Lieberman. Metafor: visualizing stories as code. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 305–307, New York, NY, USA, 2005. ACM Press.

[11] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond aop: toward naturalistic programming. *SIGPLAN Not.*, 38(12):34–43, 2003.

[12] B. A. Myers, J. F. Pane, and A. Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, 2004.

[13] D. Price *et al.* NaturalJava: A natural language interface for programming in Java. In *Proceedings of IUI*, Short Paper/Poster/Demonstration, pages 207–211, 2000.

[14] D. Price *et. al.* Off to see the wizard: Using a "wizard of oz" study to learn how to design a spoken language interface for programming. In *Proceedings of the Frontiers in Education Conference*, November 2002.

[15] T. V. Raman. Emacspeak – direct speech access. In *ASSETS*, pages 32–36, 1996.

[16] J. Sachs. Recognition memory for syntactic and semantic aspects of connected discourse. *Perception and Psychophysics*, 2, 1967.

[17] S. Shaik, R. Corvin, R. Sudarsan, F. Javed, Q. Ijaz, S. Roychoudhury, J. Gray, and B. R. Bryant. Speechclipse: an eclipse speech plug-in. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 84–88, 2003.

[18] A. C. Smith, J. M. Francioni, and S. D. Matzek. A java programming tool for students with visual disabilities. In *Fourth Annual ACM Conference on Assistive Technologies*, pages 142–148. ACM, 2000.

[19] L. Snell. An investigation into programming by voice and development of a toolkit for writing voice-controlled applications. M.eng. report, Imperial College of Science, Technology and Medicine, London, June 2000.

[20] M. Tomita. *Efficient Parsing for Natural Language — A Fast Algorithm for Practical Systems*. Int. Series in Engineering and Computer Science. Kluwer, Hingham, MA, 1986.

[21] Voice Coders. *VoiceCode: Program By Voice Toolkit*. http://www.codevox.com/pbvkit.