Spoken Language Support for Software Development

Andrew Begel

4th December 2000

Abstract

Programming environments today have not changed very much in the past thirty years. Programmers still interact with their environment by reading software artifacts on the screen, typing text with a keyboard, and moving or drawing entities with a mouse. But, when programmers interact with each other, they use their voice, draw diagrams, and make presentations. While the future of computing is moving towards more capable, higher bandwidth interfaces, such as speech, gestures and drawing, none of these modes are currently well supported by any software development environment.

In this dissertation, we propose that in addition to ubiquitous plain text editing, enabling programmers to dictate, compose, navigate, browse, and edit their software both verbally and in high-level linguistic fashion will enhance programmer productivity and make the software development process more efficient. To better understand how programmers might communicate with one another, we will conduct user studies covering basic software development tasks. Informed by this research, we will develop a set of programming language dialects to support these activities, and build a programming environment that integrates voice recognition with the programming language analysis required to understand such high-level linguistic forms of input. We will evaluate this environment in a series of user studies to better understand how novice and expert programmers communicate their ideas at a high-level, and investigate how working in this fashion can enable people learn to program more easily and help them to be more productive and efficient software developers.

1 Introduction

Virtually all software development is conducted in an interactive setting in which developers use relatively powerful personal computers or workstations, shared, networked repositories, conventionalized processes, and a diverse collection of tools and online services. There is a long history of research in support of the development activity, ranging from the study of particular languages, tools, and processes, to the design of integrated environments or tool suites. Examples of environments include programming environments [4, 48, 9, 57, 58] (in which the emphasis is on the programming aspects of software development), CASE tools (in which tools are devised for various aspects of the software life-cycle), and software development environments [30, 20], in which the integration of rich collections of tools, processes, and methodologies are envisioned throughout the life-cycle.

Most of these software engineering environments have an underlying model of a user interacting with the environment by reading software or natural language artifacts on the screen, and by typing text, or moving or drawing entities with a mouse. The future of computing and programming is moving towards more capable, higher bandwidth interfaces, where the human has many more ways to give input to the computer. Not only does this future involve multiple modes of input, such as speech, drawing, and gesturing, but these alternate modes also afford higher levels of human/machine interactions, for example, through the expression of informal pseudocode and diagram drawing.

You might think that very few of these rich forms of input have made it into software developers’ daily lives. And for the actual programming task, you’d be right. But software developers communicate with each other all the time using voice, diagrams, presentations, etc. This is the way that teachers communicate with students, and how students communicate with each other when they are working on a software project. Unfortunately, however expressive these interactions might be, none of them are understood by current software development environments. These environments support text entry into a text editor and
batch compilation services. Some (very few) environments support slightly higher-level services, like online syntactic and semantic analysis, that developers can use to explore their source code artifact with more effectiveness than "grep". But program entry is still consigned to text editing.

At this point, it may seem that we’re implying that editing source code textually is a bad thing. That isn’t what we’re saying. We believe something that’s been used for fifty years has to have some great merit to survive this long (which is why the eBook has not replaced the paper book, and why the “paperless office” has not eliminated paper). Instead, what we advocate, is that in addition to plain text editing, enabling programmers to dictate, compose, navigate, browse and edit their software in high-level linguistic terms can enhance programmer productivity and make them more efficient. Moreover, we believe that if the programmer could express himself verbally, he might find it easier to speak in pseudocode, or in some stylized, formalized – yet informal with respect to an actual programming language – high-level language, and get his ideas down on screen more efficiently.

What do we mean by the terms program dictation, composition, navigation and browsing, and editing? Program dictation is the direct entry of program text (either by speech, keyboard, or pen) in the grammar of the target programming language. Program composition, which includes program dictation, enables programmers to express themselves in a form of high-level pseudocode (such as “repeat this 10 times” to get a loop in the target language, or “create an integer named foo” to declare a new variable named foo of type int). Program navigation and browsing, which also includes program dictation, adds a command language for expressing structural and semantic-based searches through the code base (such as “find the function that converts a List into a Tree” and “show me all of the functions that call method fooBar”). Finally, program editing, which includes all of the preceding forms of input, adds commands for modifying code, as well as expressing code transformations (such “swap the first and second arguments in method mooCow”).

In computer science education, we think that this high-level verbalized approach to programming can ease learning for new students. Anecdotal experience suggests that when students talk about programs to one another, they talk in terms of constructs (e.g. methods, if-statements, classes) and semantic properties (scope, type) rather than textual entities. However, when they write programs down, they must translate their conceptual programming thoughts into the idiomatic low-level syntax of the programming language they are using. It is commonly known that most students learning to program for the first time have a lot of trouble learning syntax. It takes them quite a long time to sort out the square brackets from the curly braces and figure out exactly where the parentheses belong [13]. This has been such a pervasive problem that some educational researchers have long advocated syntax-directed editing to ease the pain of learning syntax [53, 21, 23]. This approach to high-level program entry, syntax-directed editing, enables students to edit a higher-level representation of their code, rather than edit purely in text. Since most of the confusing syntax of a language exists solely to allow the parser to recover the structure of the code, syntax-directed editing allows the editor to dispense with all of the now useless punctuation. Essentially, students pick language constructs from a menu to insert into their code. For example, I could pick an \texttt{if statement} template from a menu, and get \texttt{if (boolean expression) then statement else statement}, where I could then click on the italicized words to fill in the blanks. Syntax-directed editing can be a great tool for a novice, as long as great care is taken to ensure that as a novice gains knowledge and turns into an expert, the tool does not turn from a boon into a burden. Early syntax-directed editors were left on the wayside because their advantage in making syntax easier turned out to be a disadvantage when the system overly prescribed how proper coding should be done [40, 46, 52].

While students have trouble learning syntax, they have a much more difficult time understanding semantics. One symptom of this shows that when students learn to program in one language, the programming skills they learn don’t transfer well to other languages that they may learn afterwards [8]. At MIT, we called this “learning the zen of programming.” – that any idea can be expressed in any programming language – and it was taught in the first computer science course, 6.001. Unfortunately, I think only 30 to 40 of my friends picked up this “zen” in the first course, though everyone got it by their junior year. This isn’t taught in Berkeley’s first course on programming, though students also seem to be able to pick it up by their junior or senior year.

We envision a system that allows students to not only program at a low-level in their source language, but enables them to directly express their conceptual ideas in a formalized pseudocode. This pseudocode could be “translated” by our editor into code in the source language, and viewed either in the original pseudocode
or in translated form.\(^1\) This expression would ideally take place verbally (but we will support text-based entry as well) in order to dovetail nicely with the natural ways that students already express themselves to others when discussing software projects on which they are working.

Programming verbally can also help another community of developers, those with RSI (repetitive strain injury). Increasingly, reports about RSI in the workplace have shown that the typewriter/keyboard is contributing to the injury rate in the programming community.\(^2\) Once a developer shows signs of RSI, their productivity goes down due to their inability to continue to use a keyboard. Not only is the typing activity repetitive, but the actions that the developers take in their text editors is quite repetitive. We will present examples of these actions later on. The ergonomics community has developed new products to support typing without aggravation of RSI, but these are merely delaying the inevitable. Programmers need to avoid the use of the keyboard, and without an efficient alternative, a diagnosis of RSI can be a death knell for a programmer’s career. By using their voice, developers can save their hands.

The main thrust of this dissertation is to build a software development system that can understand spoken program dictation, composition, navigation and browsing, and editing, which will make these tasks easier for the software developer. The advent of affordable, powerful, commercial speech recognition tools has come along at the right time.

First, however, one must ask the question, how easy is it to program verbally with speech recognition systems? If they don’t provide a convenient and efficient solution for software engineering, no developer would ever use them. In order to answer these questions, we will conduct many user studies to discover how students and experts express themselves verbally during the software development process, both alone and with others. We have already undertaken one study concerning program dictation in Java, which we will discuss later in this document. We will use the results from these user studies to design a more easily spoken version of the Java programming language, as well as design the higher-level composition, navigation and editing commands that most naturally formalize how students already verbalize these tasks.

In order to enable the computer to understand these new formal languages that we will design, we will need to develop advanced programming language analysis techniques. We will build these analysis techniques on top of Harmonia, our programming language analysis framework which provides language-based tools and services to applications. We believe that we understand the extensions necessary and will argue subsequently that they are feasible to build and will be adequate to perform the task.

Finally, we will evaluate our system to see how it is used by students and expert programmers, and determine if it improves programmer efficiency and productivity.

In the following proposal, we will demonstrate the current state-of-the-art in speech recognition technology and show how current uses of these tools for programming provides inadequate support for efficient software development. Then, we explore how people would naturally dictate code (without the constraint of requiring computer understanding of their utterances) and discuss the consequences the results have on programming language design. After this, we introduce Spoken Java, a dialect of Java we are creating as the testbed for our work. We then show how Harmonia’s programming language analysis technology can be used to handle the ambiguities arising from spoken software development and indicate what enhancements we need to add to the framework to support speech. While describing the prototype editing system that we will create, we discuss our vision for verbal ways to perform program dictation, composition, navigation and browsing, and editing. In the next section, we introduce some important cognitive issues that will affect programming by voice. We propose several user studies and evaluation metrics in using this system with college computer science students. Finally, we summarize the contributions of this dissertation work, present a timeline for its completion, speculate on future work beyond this dissertation and conclude.

## 2 Speech Recognition

Fast and accurate speech recognition has been a goal in computer science for 50 years. Early techniques tried to model how the brain understands sound, speech and language, and embed this understanding into an audio recognizer. These approaches met with failure, both because understanding of how the brain

\(^1\)This translation will be supported by research in the Harmonia project that is outside the scope of this dissertation.

\(^2\)This is affecting programmers in our own Harmonia project!
understands speech was (and still is) largely unknown, and because the available computer processing power
was quite small. More recent efforts have abandoned cognitive understanding for statistical methods, hidden
Markov models, bayesian inference algorithms, context-free parsing, neural nets, and a better understanding
of the first level of sound processing by the inner ear and the brain. Combined with massive advances in
computing power, brute force algorithms, and incredibly large sound sample dictionaries, speech recognition
today is a fairly accurate and feasible solution for average desktop computers.

Commercial speech recognition engines are constructed by first recording tens of thousands of words,
spoken by people with all sorts of accents, into a large phoneme dictionary. This dictionary, combined with
an n-gram analysis of typically spoken documents, is analyzed and used to form the basis of a hidden Markov
model or neural net which does the actual recognition. To make translation between phonemes and words
quicker, speech recognizers use language grammars (both specified for the application and for the natural
language) to prune their search.

The supported grammars are divided into two categories: command and control and dictation. Command
and control grammars are fixed grammars (meaning all words, or terminals, are known at grammar compile-
time) which are used to script applications. As each production in the grammar is recognized, an action
is performed, such as “Open File Menu”. Dictation grammars, on the other hand, are defined specifically
for each natural language. IBM’s ViaVoice [27] and Dragon’s NaturallySpeaking [17] are two examples of
speech recognizers that support dictation in English (IBM’s also supports many other languages including
Spanish, German, French, Chinese, and Japanese). Knowledge of the grammar of the natural language helps
the tool to disambiguate homophones, \(^3\) discern proper capitalization, and help make the speech recognizer
run faster.

Efforts to apply speech-to-text conversion for programming tasks such as authoring, navigation, and
modification using these conventional natural language processing tools have had limited success. English
language parsing provides poor recognition of most traditional programming language text because the
grammars are structurally very dissimilar. Thus, practical efforts to use speech recognition for program
dictation have used command and control grammars. Since these grammars require a fixed set of terminals,
and generally don’t scale very well, complete specification of a programming language has not been possible
(nor tried). Instead, developers apply \textit{ad hoc} techniques to script the speech recognizer.

VoiceGrip [16] uses Perl and an extensive set of macros to match regular expressions to uttered program
text and produce program code in the source language. Code dictation is awkward and over-stylized. For
example, the following illustrates how you must speak to enter a loop to repeat a body of code ten times:

\(^3\)Homophones are words with different meanings that sound alike, but may be spelled differently. In contrast, homonyms
are words with different meaning that sound alike are spelled the same.
### User Speech | Computer Output | Commentary
--- | --- | ---
for statement | for( ; ; ) { } | Inserts a for statement template
next | for( ; ; ) { } | Moves the cursor to the first slot
declare variable name india | for( i ; ; ) { } | Inserts new variable i
variable type integer | for( int i ; ; ) { } | Modifies variable to add type annotation. “integer” is mapped to int.
assign zero | for( int i = 0 ; ; ) { } | Initialize i to 0
next | for( int i = 0 ; ; ) { } | Moves cursor to next slot
recall one | for( int i = 0 ; i ; ) { } | When first defined, identifiers are stored in a cache pad for later retrieval by number.
less than ten | for( int i = 0 ; i < 10; ) { } | Indicate the loop limit.
next | for( int i = 0 ; i < 10 ; ) { } | Move cursor to next slot.
recall one | for( int i = 0 ; i < 10 ; i ) { } | Insert loop variable
auto-increment | for( int i = 0 ; i < 10 ; i++ ) { } | Add in increment by one operation
next | for( int i = 0 ; i < 10 ; i++ ) { } | Move cursor to for loop body
... | for( int i = 0 ; i < 10 ; i++ ) { ... } | Loop body code entry
end for | for( int i = 0 ; i < 10 ; i++ ) { ... } | Move the cursor past the end of the for loop.

As you might guess, this kind of code entry is very slow.

In addition, the regular expressions used by VoiceGrip are inserted into an editor that knows nothing of the program structure, limiting editing and navigation to word processing services or those based on ad hoc tokenization of the text buffer. Many users of this tool, mainly RSI sufferers, must resort to using the keyboard to navigate through and edit code with any speed. Such users often create an extensive set of editor macros to help spare their hands further harm.

Recently, there has been an effort to unify VoiceGrip and several other speech-for-programming efforts into a program by voice (PBV) toolkit [60]. The toolkit incorporates an English language voice recognition mode interface for Emacs and several packages of macros for easier navigation of source code by voice using line numbers and regular expressions. This provides some improvement, but is still far short of an ideal interface. In addition, new efforts at Drexel University [2] and Compaq SRC have started to work on this problem, but these projects are still in the design phase and have no results to show.

### 3 How do people naturally speak code?

Programming languages have primarily been designed as a written medium, which suits the traditional mode of input, the keyboard, very well. Programmers spend years learning how to “write” in these languages, translating the ideas in their heads into code on paper or on the screen. The current state-of-the-art in programming by voice only provides services for program dictation, and those services are quite awkward. What would an ideal vocal interface for program dictation be like? What about for other services, such as program composition, navigation, browsing and editing? We conducted an experiment to find out how people most naturally speak source code to gain some insight into the program dictation process.

#### 3.1 The Experiment

To understand how programming dictation might change in the presence of vocal interaction, we designed an experiment to have participants read a one-page pre-existing Java program out loud. Each participant was told to read the code as if he were programming with a partner on a group project. The participant
was told that his partner was sitting at the keyboard, and the participant had to get the gist of the code across to his partner by reading it out loud. His imaginary partner was assumed to be conversant in Java, programming, and the project at hand. The experiment was intended to give some insight into program dictation.

We carried out this experiment with ten computer science graduate students, five of whom were familiar with Java, and five of whom were not, but all of whom were programming-literate in some computer language. We audio-taped all ten students, and transcribed their recitations onto paper, keeping all utterances and performing little semantic transformation (e.g., numbers are written as they are pronounced, one fifty eight, rather than 158, but “one” is disambiguated from “won” and “eight” is disambiguated from “ate”).

The results of this experiment are instructive. We organize them into three major categories: Java language design, vocal program dictation, and vocal program composition.

- **Java language design**
  - Awkwardness by design
    * Some Java language constructs are awkward in English. An example is casting objects to a type. In Java, the cast comes before the type – `(int)foo` – but in English, the noun needs to come before the modifying verb, so people resorted to creating pronouns - “cast an object to int, that object is foo” Some speakers reversed the order of the statement - “cast foo to int”.
  - Individual variability
    * Students are inconsistent when they speak particular constructs. For example, `System.out.println` is sometimes said “system dot out dot print line”, and sometimes “system out print line.” Even when the same person encountered the print statement again, he might say it in a different form.
    * Most speakers verbalize most constructs the same way, but there is great variability for other constructs. Examples of recurring verbalizations are array reference: `foo of i, foo sub i, element i of array foo`; and variable assignment: `initialize foo to 5, set foo to 5, foo equals 5, assign 5 to foo, allocate a new array of length 5 to foo`.
  - Ambiguous words and semantics
    * Some of the ways that people speak particular constructs are ambiguous, and often rely on non-verbal cues, or program context to disambiguate them. An example is `foo[i++]` vs. `foo[i]++`. Many people said `foo of i plus plus` for both constructs. Essentially this amounts to dropping the close bracket in the array reference. English speakers inserted a pause before the `plus plus` when saying the second statement; non-native English speakers resorted to a restatement of the entire expression (e.g. `position i of array foo and then incremented by plus plus`).
    * Some words that people say are homophones, that is, words that sound the same. An example is `to` and `two`. The listener can usually resolve this kind of ambiguity from context; a computing system will have to do so as well.
  - Native English speakers vs. non-native English speakers
    * There are differences between native and non-native English speakers. Small non-English identifiers (like “tur”, “pat”, “spp”) were spoken as syllables (word fragments) if possible by the native English speakers in our sample, and spelled out otherwise. But the non-native English speakers always spelled out all word fragments even if they were pronounceable.
    * Some words in Java are unpronounceable even by native English speakers. There were a variety of coping mechanisms: short words were spelled out, long words were converted into verbalizable words. Some programming language identifiers and keywords are shortened for ease of typing (for instance, `println`), but when readers spoke these words, they lengthened the words to their original long form (print line).

- **General program dictation**
Too few or two many words

* Students didn’t say most of the punctuation characters. Semicolons were never spoken, open and close parens as part of method calls were never spoken. Sometimes the commas separating arguments to a method call were spoken, but this was inconsistent. Methods with more arguments tended to cause people to say the comma. Dots marking qualified identifiers were sometimes spoken and sometimes skipped, though it was far more common to say them. Close curly braces were often “translated” into context-sensitive terms (“then” followed by “close the if”, a method body followed by “end method”, etc). Open brackets for array references were often translated into “of” or “sub”. Close brackets were not spoken. Quotes surrounding strings were rarely spoken, even if the string had multiple words.

* Speakers use a lot of extra words that are not part of the programming language syntax, or even its descriptive metalanguage (e.g. phrase names). This includes a lot of English-style phrasing that is more comfortable for people than sticking to a very strict programming language grammar. For example, speakers use articles (“the”, “a”, “that”) and conjunctions (“and”, “then”, “after that”).

* Speakers use lots of stop words (“uh”, “um”), and repeated words. Word repetitions are caused by a person thinking a bit and then repeating the last few words that they said in order to form a complete sentence.

Speaker Mistakes

* Speakers make and correct mistakes with a small, but definite frequency. There are usually indicator words like “wait”, “oh, shoot”, and “stop” that begin an error correction phase. However, this phase was more colloquial than something formally stated. Speakers also add their own out-of-band comments like “wow, this is fun”.

General program composition and navigation

High-level coding

* People spot patterns in the code and explain the pattern rather than speaking the code word for word. For example, “there’s a comment here, and this is what’s inside...”, “declare the constructor”, “now let’s initialize a bunch of foo’s fields”, “we need to catch these exceptions, foo and bar”, “enlarge this pool array by twice its length”, “catch the exception and if that occurs print the stack trace”, “this method takes no arguments”.

* Although this experiment was about sequential program dictation, speakers used navigation keywords to move the “cursor” around. Examples include “and the next method is”, “close method”, “close that block”, “and then you’re going to have the constructor”, “start with package starlogo”.

3.2 Observations from the experiment

First, the experiment has several flaws. The students did not write the program that they were asked to read. The students had about two minutes to familiarize themselves with the code. The students were reading from a page of pre-written code, rather than composing code on the fly. There was no visual or audio feedback in response to code dictation. Some of the users’ utterances might be due to lack of knowledge of where the cursor was, or whether or not the current block of code was closed yet. Some students were unsure what style they should use to speak the code. Some of them insisted on speaking all punctuation, as if they were speaking into a keyboard, rather than to another “person” (which were the instructions they were told).

However, despite its limitations, the experiment points towards some of the capabilities we will need. We propose to conduct additional experiments to guide our work. For example, this experiment deals only with dictation, a component of authoring. We need other experiments to learn what capabilities are most valuable for high-level program composition, navigation and editing.
3.3 Spoken Java

Using the results from our experiment, we are creating a dialect of Java, called Spoken Java, with altered syntax to make it more natural to speak. Why not develop a new language from scratch? Some researchers studying programming for end-users believe new paradigms might help programmers express themselves better. Natural Programming [39] is one of several radical approaches to programming from an HCI point of view. It advocates reinventing the programming language in the style of how a child views process and control. While this idea is interesting and may turn out to be fruitful, we feel that designing a programming environment around an existing language is easier to integrate into current curricula. We don’t want to teach students how to program from scratch; we want to make their existing job programming easier.

So why choose Java? We picked Java because it is widely-used in education (at Berkeley in CS61b and CS164), as well as in the corporate world, so our proof-of-concept will be readily appreciated. In addition, there are many externally developed Java analysis tools that we may plug into our HARMONIA language analysis framework. Notice, also, that we only change the syntax of Java, not its semantics, because we know that semantics are hard to learn (and hard to design correctly). If a student learns Java, they can directly apply that knowledge outside of their classes.

The design of the Spoken Java grammar omits much of the punctuation that makes the traditional Java syntax easy to parse. In fact, while the original Java grammar is completely unambiguous, Spoken Java currently has 400 ambiguities! How will we understand what code the user is saying if there are so many ambiguities?

To solve this problem, we will build on our HARMONIA language analysis technology. The marriage of 16 years of research in incremental lexical, syntactic and semantic analysis for programming languages will help make vocal program dictation recognition feasible and efficient.

4 HARMONIA

Our research group has conducted research on tools and development environments over a long period. Its research interest focuses on providing the underlying technology that makes integrated software development both possible and powerful. In our three major projects on interactive development technologies, our emphasis has been on the importance of language-based support, and on exploiting the advantages of persistent, interactive, presentation-rich settings. The Pan environment [7] was created to explore topics in language-based editing. It yielded new results in incremental syntax and semantic analysis [5, 6] and in user-centered language-based services [55]. The Ensemble project [24, 25, 62, 64, 66] studied multiple-representation language based document manipulation, for both formal and natural languages.

Our current project, HARMONIA, is ideally suited for supporting the necessary programming language analysis. The HARMONIA framework was derived by embedding some of the software built for the Ensemble project in a new and extensible architecture. While Ensemble’s novel direction was to explore the combination of (improved) interactive language-based analysis and multi-media document structuring and presentation, HARMONIA focuses on providing a flexible set of embeddable tools for applications that require incremental, language-based analysis services.

Incrementality is fundamental to the HARMONIA framework because it is the basis for building scalable, interactive environments. The amount of re-analysis is proportional to the amount of change, not the size of the entity that has been changed. This incrementality will be essential to realize practical analysis for processing voice input.

The heart of the HARMONIA framework is the Language Kernel, which provides the infrastructure for linguistic analysis and manipulation. The centerpiece of the Language Kernel is the annotated (or attributed), syntax tree abstraction, which constitutes an internal representation of program code or other source documents in the HARMONIA framework. A distinguishing feature of the HARMONIA syntax trees is the explicit representation of whitespace and comments. In addition to capturing user-provided formatting and annotations, such representation allows the Language Kernel to “unparse” the syntax tree, generating a textual

---

4The languages handled by the Language Kernel are primarily those that have a syntactic and semantic structure akin to programming languages. Since there is no generally-accepted word for artifacts written in those languages, we use the term “programs”. The context should indicate when the discussion pertains only to programming languages.
representation of the program that may be required by some text-based tools [65].

Harmonia annotated syntax trees are modeled as self-versioned documents, constructed from versioned primitive data types [63]. In other words, the document representation incorporates a fine-grained history of changes to it. Self-versioned documents facilitate access to previous versions by transparently incorporating modifications as they occur. Not only does that support powerful forms of undo operations, it is an underlying mechanism for maintaining a development history. Because the versioning is organized structurally rather than temporally, changes to particular portions of the program can be recovered even if other changes were made in between. Additionally, versioned documents provide an efficient change-reporting framework that can be utilized by various tools (for example, providing diagnostics in terms of differences from past versions, rather than guesses at the nature of misuse of language rules). The history mechanism is sufficiently general to capture changes made to attributes as well as to syntax.

This history mechanism will be vital to supporting the kind of feedback and rollback support required for processing possibly erroneously processed vocal input. There are many occasions for the users' intent to be misrecognized or misanalyzed, resulting in potentially harmful changes to their program. In such a noisy environment one must either support undo, or else ask the user to confirm all changes before they happen. Since asking the user prior to making every change will be prohibitively annoying, we will use Harmonia's history mechanism to support the powerful forms of undo which will be required.

The Language Kernel provides incremental syntax analysis that can both construct the syntax trees from traditional text files, and incorporate changes to the syntax trees incrementally as they are introduced. The Language Kernel includes two incremental lexer drivers and two incremental parser drivers. One parser driver is based on LR-parsing technology [66]; the other supports incremental Generalized LR parsing (GLR) [64, 47, 54]. The descriptive power of GLR eliminates the need for most "grammar-hacking" and allows a syntax specification that naturally corresponds to abstract syntax, without the need for complex mappings [5, 56].

GLR is a context-free parsing algorithm originally derived from Earley's parsing algorithm [18]. Unlike a more traditional parsing algorithm such as LL(1) or LALR(1), GLR supports ambiguous context-free parsing. Originally designed for parsing natural languages, GLR has proven valuable in simplifying programming language grammars to the point where the grammar resembles the ideal abstract syntax. This freedom comes at the price of introducing ambiguities into the parsing process. but for a language originally designed to be conflict-free, the conflicts can be eliminated once the program has been fully parsed. When the grammar is inherently ambiguous, the final result will not be a parse tree, but a parse forest. First introduced by Tomita, a shared-packed parse forest is a data structure capable of recording all possible parse trees produced by the GLR parser in only slightly larger than $O(n)$ space.

The GLR algorithm is an important part of our solution to resolving ambiguities in vocal expression of the various actions involved in software development. It can be thought of as a slightly more complicated variant of LR parsing. As with LR parsing, a parse table is constructed from LR(0) item sets. However, when multiple actions are found to insert into a cell in the table (indicating a shift-reduce, or reduce-reduce conflict), LR table construction algorithms choose one action to insert. In contrast, GLR table construction algorithms place all of the actions into the table's cell.

As with LR parsers, GLR parsers use a parse table, a parse stack, and an input token stream to produce a parse of the input. To optimize space during the algorithm's execution, the parse stack is actually represented as a graph, to better handle the multiple contexts necessary to record multiple parses at the same time. At runtime, GLR parsing acts identically to LR parsing when processing productions that have no inherent ambiguity. However, when both a shift and reduce action are called for by the parse table, GLR forks the parsing process. Essentially, it splits the current parse stack into $n$ parse stacks that each run one of the conflicting actions. Since the parse stack is really a graph, we merely add an extra edge to the graph and record that there are multiple active parsers. Each active parser is able to shift the current token, but by default, if there is an active parser that is able to take a reduce action, it is executed first. This enables all of the active parsers to follow the input token stream together. If a parse stack fails to parse the input correctly, it is declared inactive and removed from the parse graph. If two parse stacks ever reach the same state and are both ready to consume the same input token, they are merged. This allows the parsing algorithm to cut down on duplicated work and isolate the ambiguous piece of the parse tree to the portions of the input that are ambiguous. When the parse graph is reduced to the start state and all of the input has been consumed, the result is a shared-packed parse forest which contains all of the possible parse trees that
had been identified by the successful active parsers.

Our GLR parser is slightly more elaborate than what I’ve described here because operates incrementally. If there are changes in the input token stream, at any place in the stream, the parser can reparse the changed tokens without deconstructing and reconstructing the entire parse forest. This incrementality is crucial to supporting vocal input in real-time.

In addition to lexing and parsing, the Language Kernel includes the infrastructure for semantic analysis, which can be utilized by a language description writer to provide semantic analysis services. Our current implementation of semantic analysis support is not incremental and does not allow analysis across multiple translation units. As discussed in a subsequent section, we are extending this infrastructure to provide fully incremental semantic analysis through an attribute grammar evaluator.

In order to give the developer needed flexibility in modifying programs, the Language Kernel continues to provide services when programs are ill-formed, incomplete, or inconsistent [55]. The incrementality of the HARMONIA parsing algorithms, together with history-sensitive error recovery [61], naturally incorporate inconsistency into the syntax tree by enabling syntax analysis to continue beyond malformed regions, and by enabling malformed regions to contain well-formed substructure. This capability is required to support the propensity of developers to jump around their code at will while editing, leaving many code artifacts incomplete. Incompleteness is handled transparently by the framework.

HARMONIA’s language-specific information is carried by language modules. A language module may encompass both hand-coded mechanisms and information that is derived from a formal specification. Each language specification is translated into a set of C++ files, which are then compiled and linked to produce a shared (dynamically linked) library. This compilation is performed off-line, permitting optimizations that result in time- and space-efficient representations and algorithms. Any programming language having formal syntactic and semantic specifications can be easily added to the HARMONIA framework by providing a syntactic and semantic description of that language. The language module architecture is open-ended, allowing each language to provide other kinds of analyses and services (e.g., control/data flow, program transformations). It is also possible to interface external tools as part of the analysis process.

The shared libraries representing compiled language objects are loaded on demand into the running environment. This arrangement makes it possible to support a large number of languages simultaneously without the environment itself becoming unmanageably large, and precludes the need to specify the set of available languages when the environment is built. 5

Even though the HARMONIA framework is written in the C++ programming language, it is clear that C++ is frequently not the most appropriate language for building interactive tools. To facilitate rapid development of such tools, we have created bindings for higher-level programming languages including Java, Tcl, and Emacs Lisp. The latter lets us utilize a widely-used editor rather than creating a legacy editing front-end. We have also developed an interface to XEmacs [12] that gives the user access to HARMONIA services as well as the usual XEmacs services. When we are ready to do further work on advanced presentation and editing models, we will determine whether that editor can be enhanced appropriately, or whether we need an alternative editing interface with support for a different user model.

5 Mixing Voice and Programming

How can HARMONIA help in constructing a system to understand voice-based programming? In this section we show why support for incrementality and ambiguity in the analysis process is vitally important to understand spoken program text.

We stated earlier that programming languages grammars have traditionally been designed to be easy to parse—thus, all ambiguities are eliminated when the grammar is compiled. This requires adding extraneous productions to the grammar, and usually results in superfluous concrete syntax visible to the programmer. Typical programming languages use delimiters such as punctuation and white space to facilitate that non-ambiguity. We conjecture that this obfuscation is mainly the result of programming language’s text-based origins, and not due to intentional ignorance of syntax usability metrics.

5 Thus HARMONIA is not a ‘template-based’ monolingual framework as are the environments produced by the Synthesizer Generator [48] and other similar systems.
The foremost design goal for a vocal programming language and environment is different – the language must be easy to speak. Not only will we need to create vocal dialects of existing languages, but we also have to create stylized languages for program composition via pseudocode, and program navigation and editing via command languages. Some of the contributions in this dissertation will be the design and evaluation of this set of vocal programming languages applied to the Spoken Java program dictation language.

We plan to use the GLR parser from our HARMONIA framework to parse and understand these languages, but as we saw from our dictation experiment, human speech is a little more ambiguous than the traditional text-based input for which HARMONIA was designed. In the next section we describe the additional ambiguities that arise from spoken language input and show how we will extend the various components in HARMONIA to support them.

5.1 Types of Ambiguities

As user interface designers have developed drawing, gesture and speech interfaces, imperfect recognition technologies have created a need to disambiguate user input. Recognizers are built to pass ambiguities in recognition to their clients, which must then use syntactic and semantic knowledge to resolve them, or else present the ambiguity to the end-user. Various techniques, such as repetition detection and resolution (when the user repeats inputs) [43, 32, 37], choice detection and resolution (when two interpretations are equally likely) [37, 1, 17, 28], automatic disambiguation through rules [3], historical statistics [38] and multi-modal interference [42, 51], have been developed to reduce recognition error rates.

In the programming language domain, we apply these techniques in a formalized way to resolve ambiguities at each stage of analysis. There are three kinds of ambiguities: lexical, syntactic and semantic. Lexical ambiguities occur at the word level, syntactic ambiguities affect the structure of what is recognized, and semantic ambiguities affect the meaning of the code.

5.1.1 Lexical Ambiguities

There are several lexical ambiguities found in speech. The first category is what I call one token, many spellings. The spoken voice cannot convey the capitalization which is important in many programming languages. In addition, when the speech recognizer returns a word, we may have actually said any number of homophones of that word. For instance, if the speech recognizer returned “one”, how can we know that it was not “won” that we said? If we enhanced the token data structure to allow alternate spellings, we could pass all alternatives to the parser.

Another category of lexical ambiguity is alternate tokens for the same input word. This is not due to homophones, but in fact to the language defining the same word in multiple lexical categories. Explicitly stated punctuation falls into this category. In Java, “.” is used to separate identifiers and indicate field reference. Likewise, someone may use “period” as an identifier for a variable. In a textual representation, these two are easily distinguishable, but in a verbal setting, we can’t tell the difference. Thus, in order to support both uses, we must pass two alternate tokens to the parser for this single input word. This requires changing the lexer-parser interface to allow more than one token to be passed when the “next” token is requested.

A third very important category of ambiguities occurs because the speech recognizer can not regulate whitespace to the degree that a text editor can. If there is a multi-word identifier (such as “printLine”), we would say two words, “print” followed by “line”. However, the speech recognizer will insert whitespace between every word, even though in this case, that is not wanted. Our solution for this ambiguity involves creating every possible concatenation of tokens as single tokens for the parser. For example, “foo bar moo” would be sent as four alternate token streams: “foo bar moo”, “foobarmoo”, “foo bar moo”, and “foobarmoo”. Now, lest we accidentally force the lexer to construct the power set of input tokens, we can impose two constraints: first, only identifiers may be concatenated together (we can design the language keywords to eliminate multi-word tokens), and second, we can bound the number of adjacent concatenated tokens based on the natural language of the speaker. In English, there are few identifiers that consist of more than five words concatenated together. On the other hand, in a language like German, there might be a different limit. Each of these, combined with aggressive automatic ambiguity resolution in the parser and semantic analysis, will enable us to bound the amount of lexical ambiguity this fact causes.
The last ambiguity concerns misrecognized tokens. Speech recognizers have trouble with partial words and unpronounceable words, since they are not found in its dictionaries. However, these are important because legacy software often uses shorthand words for identifiers to ease typing. Speech recognition systems come with pronunciation feedback tools which recognize unpronounceable words and ask the user to say them. Thereafter, when the user utters that sound again, the speech recognizer will output the prechosen spelling of the word.

5.1.2 Syntactic Ambiguities

As the lexer passes tokens to the GLR parser, the parser must try to put them into a parse tree. But, as we said before, verbalizable language grammars will have inherent ambiguity, especially if we design the grammar to minimize spoken punctuation. For example, the parentheses marking method invocation may not be spoken. The “dot” separating field references may be omitted. Semicolons as end of statement indicators may be omitted as well.

Our incremental GLR parser must first be enhanced to accept the various changes we’ve made to the lexer-parser interface. In GLR, whenever a conflict is detected (e.g. shift-reduce or reduce-reduce), the parser that faced the conflict is split into two, and each parser takes one of the choices. Allowing alternate lexemes for the same input word will introduce a additional shift-shift conflict for the given parser. The logic to handle this is easily added to the parser algorithm.

A more complicated solution is required to support multiple token streams for each parser. If all possible concatenations are generated, then each parser may not be at the same point in the token stream as each other, which would violate a precondition of the GLR algorithm. We need to elaborate the interface between the parser and token stream to give each parser its own pointer into the stream. We must be careful, however, not to break the parser merging component of GLR. This merging allows two parsers to merge if they end up in the same state and face the same input token. We must allow merging even if the parsers read two different token streams.

Spontaneous speech also contains extraneous words, which are not part of the language grammar. Without any filtration, these extra “stop” words will confound the parser and cause it to reject otherwise legitimate parses. We plan to adapt a solution used in the GLR* algorithm that enables the GLR parser to skip words and construct a parse forest with the parses that have the fewest skipped words [33]. This is done by allowing shift actions to be performed by inactive parse states. Heuristics must be used to ensure that the parser is not forced to parse the set of all subsets of tokens found in the input stream.

5.1.3 Semantic Ambiguities

A natural language speech recognizer uses the natural language grammar and semantics to disambiguate homophones, parts of speech and capitalization. Fortunately, disambiguation of programming languages will be considerably easier than natural language because the context provided by the program will be a rich source of disambiguating information.

There are two aspects to the semantic analysis of programs or similar linguistic documents. The first is akin to the semantic phase of a compiler, namely, binding analysis, name resolution, type checking, and other analyses derived from the semantic definition of the language. We refer to that as semantic analysis. The second determines properties of the particular instance, e.g. the program. Examples of such analyses include data flow, alias analysis, complexity measures, slicing, etc. We refer to that as program analysis.

In the Pan system, we developed an incrementally evaluated structured variant of logic programming to define and maintain semantic consistency and constraints on formally specified language documents [6]. That was an elegant solution, with built-in query support, but suffered from significant performance problems. Ensemble’s implementation of semantic analysis is based on the Colander II system [35], which transforms an attribute grammar-based description of the analysis into a compiled incremental attribute evaluator that is applied to the syntax tree whenever semantic information is requested.

Colander II has both strengths and weaknesses. Among its strengths, the metalanguage is very expressive, while also exposing incrementality rather than hiding it in semantic functions. The evaluators handle long-distance dependencies and aggregate attributes efficiently, and exhibit excellent overall incremental performance. The incremental evaluation methods have more general applicability than just for attribute
grammars. However, the formalism shares with most attribute grammar systems [15] the property of being monolithic, in the sense that all aspects of the specification potentially interact. That makes the avoidance of circularity difficult, despite the more powerful mechanism we have developed [35], and inhibits the ability to encapsulate “chunks” of functionality, such as symbol table abstractions, for re-use in other language specifications. For example, the ability to specify a multi-phase analysis, instead of having evaluation order determined by dependencies alone, would simplify the design of the specifications and facilitate specification re-use.

When the Harmonia framework was created, we removed Colander II support, and only replaced it with an ad hoc mechanism for semantic analysis of a parse tree that is custom-written in C++. This solution is inadequate for the kinds of disambiguation analysis we will need. As part of our research we will explore a hybrid approach of Colander and Colander II that lets the analysis writer split the analysis into multiple phases, with a well-defined interface. The different analysis components can then use different algorithms, some incremental and some not, to achieve a better balance between ease of specification and efficient incremental evaluation. The choice of analysis method would be tuned to such factors as the locality of dependences or the utility of change-based information. At the coarser grain of the interaction of analyzers, the analysis would remain incremental.

In addition, the new analyzer must support set-based analysis of a parse forest with the ability to prune “incorrect” subtrees from the forest if found semantically inconsistent. In the natural language community, where GLR is one of many popular parsing techniques that produce parse forests, learning how to perform semantic disambiguation on the compressed parse forest data structure has become an increasingly important area of research [41, 49, 14, 26, 59]. Due to the much larger ambiguities in natural language syntax, expansion of parse forests prior to traditional semantic analysis on each parse tree would be infeasible due to the space required. We believe that a set-based attribute grammar would be able to operate efficiently on the parse forests produced by Harmonia and will adapt techniques from the natural language community for our uses.

Finally, we require support for semantic queries in the Harmonia framework. Semantic queries are used by program navigation services in order to find constructs in the user’s program. In addition, queries are also useful for incremental disambiguation. For instance, when the user is up to a point in his input where he may enter an expression, we may wish to identify what variable and method names are currently in scope. We can use this information to restrict the number of words that the speech recognizer will accept in order to achieve higher accuracy rates. This will involve querying the partial parse tree results and semantic attributes that have been analyzed up to this point and ask for this information. The semantic query functionality that will be used by this dissertation is being implemented as part of the overall Harmonia project, and not as part of this dissertation.

5.1.4 Ambiguity Resolution

This section aims to give a flavor of the form of automatic disambiguation techniques that Harmonia will be capable of. Consider the following example.

Suppose the array reference syntax in Spoken Java is

```
ArrayReference → ELEMENT Expression OF ARRAY Expression
```

Suppose the user says

```
ELEMENT S P P OF ARRAY FOO PLUS PLUS
```

The keywords ELEMENT, OF, and ARRAY serve to partially determine the structure of the input. The lexer and parser can recognize that structure, but must then parse the two expressions. There are two ambiguities: the first is the form of the expression entered between ELEMENT and OF. The second is the associativity of the PLUS PLUS operator.

Consider the first example. The lexer will determine that “S P P” could be either three single letter identifiers or fewer but longer identifiers. Using “;” as a list separator, the four possibilities are

```
s p p; sp p; s pp; spp
```

Suppose that in our Spoken Java grammar, the dot between two identifiers may be omitted (as in `foo.bar` signifying the bar field of object foo). In addition, the parentheses demarcating the beginning and end of
the arguments to a method call are optional in the grammar (because no one seems to speak them). Then
the four lexical sequences have a large set of potential parses, among them:

s p p; s(p, p); s(p(p)); s.p p; s.p(p); s p.p; s(p.p); s.p.p; sp p;
sp(p); sp.p; s pp; sp.p; s(pp); spp

However, the parser would not generate any parse having adjacent identifiers (shown as separated by
a space), since that is disallowed in expressions by Java’s grammar. Hence the list of syntactically legal
expressions includes:

s(p, p); s(p(p)); s.p(p); s(p.p); s.p.p; sp(p); sp.p; s pp; s.pp; s(pp); spp

Some of these expressions may not be valid semantically. The semantic analyzer can determine which
expressions are acceptable and which ones should be eliminated. For example, if there is no local variable,
instance field or static field “s” defined at this point in the program, then the expressions that use “s” as a
variable reference are eliminated: s.p(p), s.p.p, and s.pp. If “s” is not an in-scope method, and there is no
method in the current class named “s” then s(p, p), s(p(p)), s(p.p), and s(pp) are eliminated, leaving:

sp(p); sp.p; spp

If “sp” is neither a local variable, nor a field, nor a method, the first two choices will be eliminated,
leaving spp. If spp is neither a local variable nor a field visible from this point in the program, then an error
is reported. If all three possibilities are semantically valid, then the developer can be asked to disambiguate.

As for the other part of this example, we now have

ELEMENT SPP OF ARRAY FOO PLUS PLUS

The possible parses either increment FOO by one, or increment the result of the array reference by one.
In standard Java syntax, either (FOO++)[SPP] or FOO[SPP]++

The semantic analyzer can disambiguate this example. FOO is a Java array. In Java, you can’t increment
an array reference, thus the first option is illegal. The system therefore chooses the other possibility (which
does type check correctly since FOO, let’s say, is an array of numbers) as the correct one.

5.1.5 Manual Disambiguation

We note that not all ambiguities can disambiguated through analysis. What if there were both a field and a
method with the same name? What if two possibilities for token concatenation were both equally valid (e.g.
print("line") vs. printLine())? Capitalization affects semantics in many languages, and could have a
big impact on program structure (e.g. in Java, Class (a type) vs. class (a keyword defining the beginning
of a new class definition)).

In addition, we believe that some ambiguities should not be resolved automatically, even if they could.
For instance, the precedence of the else statement as part of a nested if-then-else clause is typically
resolved in the language grammar. Why? This is because programming language grammars were required
to be unambiguous. However, this early resolution is dangerous since the user is unaware that there might be
a problem. And, if the grammar specifies a different precedence than the one the user had in mind, choosing
the unanticipated parse tree could lead to semantic bugs in their code.

Since we’re using a GLR parser, we preserve all parses through to the semantic analysis phase. After
we’ve run our disambiguation algorithms, we will engage the software developer in a dialogue and ask them
to pick the proper parses from the ambiguities that are left in the parse forest. The chosen resolution of each
ambiguity would be stored in the resulting parse tree and saved with the program (so that the user will not
be asked again each time the program is analyzed).

What form should this dialogue between the user and analysis system take? Interactive techniques are
able to disambiguate speech and pen input before it arrives at the lexer [50], but this is undesirable because
many of the ambiguities might very well be resolved automatically through context or semantics. Current
research by Mankoff et. al. [37, 36] develops the ability to pass ambiguities through the application to
modular mediators, which can provide a semantically-aware process for ambiguity resolution. We aim to
incorporate these latter ideas into our prototype programming by voice editor. In addition, we will provide access to the standard speech recognition tools for improving spoken language recognition accuracy and for adding new words to the dictionary.

6 The Program Editor

As part of the Harmonia project, we have added harmonia-mode to the XEmacs text editor. Harmonia-mode provides an interface between Harmonia’s language services and the text editing services provided by XEmacs. We maintain a mapping between the XEmacs character-based buffer representation for a document and Harmonia’s parse tree representation. All editing takes place using character and word manipulation features found in XEmacs, which are then correlated with the node(s) in the parse tree that are affected. A reanalysis is performed on each change to the buffer (if invoked via keyboard, it is usually per-keystroke) in order to keep all linguistic information up-to-date. All lexical, structural and semantic annotations are hidden from the user until high-level linguistic services are requested. XEmacs’s ubiquitous ad hoc programming language services, which include syntax highlighting and proper code indentation, are reimplemented in terms of Harmonia’s accurate analysis information.

As part of this dissertation, we have added speech recognition services (provided by IBM’s ViaVoice) to XEmacs through its external module interface. Coordinated by XEmacs Lisp, input from the recognizer is sent to Harmonia for analysis and actions are taken based on its content.

Next, we describe how we will implement the different programming tasks in our editor.

6.1 Program Dictation

Program dictation is the one area that has been explored by others using ad hoc techniques that require over-stylized input methods [29, 16, 60]. None of these approaches discovers or preserves any syntactic or semantic structure other than any discovered by the editor itself.

Our more natural Spoken Java grammar will make it much easier for the user to code using direct program dictation. We can use our incremental analysis framework to continuously parse what the user is saying and construct a parse forest to be displayed in the editor. We will do this by intercepting the results of dictation mode prior to the speech recognizer’s own natural language semantic analysis phase, and send them to our analysis framework. Using GLR to enhance natural language speech recognition was explored on a research speech recognizer in Japan and was shown to be between 80 and 99% accurate on a task-specific grammar [31]. We would like to adapt these techniques to the ViaVoice speech recognizer, but we may have to switch to a research speech recognizer to gain proper access to the recognizer internals.

An alternate way to support direct program dictation is to continuously update a simple command and control grammar with all possible speakable lexemes at any given point in the program. Harmonia’s query support will allow us to find this out using a cursor position, the current document, and the document’s programming language grammar. For instance, in our example above, when the user said \texttt{s p p}, only identifiers were allowed there. Since the location involved a variable usage and not a definition, legal code would say that the spoken words must form a valid identifier, one that has already been defined. We can seed the command and control grammar with these identifiers when that point in the grammar is reached. This would not only make it feasible to do direct program dictation, but would also help make recognition more accurate by limiting what words the recognizer will accept.

This latter technique may have two problems, however. Many programmers do not code in linear order; they jump around. If they try to speak a identifier before it has been defined, it may not get recognized. Even so, how do they define a new identifier? Perhaps we should switch the recognizer into dictation mode when an identifier is expected as part of a grammar production, or have the user just type it in. This solution’s feasibility depends on the granularity of the programmer’s coding process. Do they jump around in the middle of statements or only at statement or structural boundaries?

Another problem with command and control grammars is performance. In current speech recognizers, changing the active command and control grammar is an expensive operation. We hope this will improve in the future, but do believe that the cost is related to the size (and maybe the complexity) of the grammar.
If we can keep the size and complexity down through aggressive ambiguity resolution, we should be able to avoid performance issues.

### 6.2 Program Composition

Supported by the Harmonia analysis framework, we can provide several forms of program composition that will better approximate how people speak code today. First, we propose to support a template instantiation feature. Each production of the grammar can be associated with a spoken keyword. When this keyword is spoken, an instance of the grammar production can be inserted into the parse tree at the current cursor location, where terminals are filled in by text, and non-terminals are filled in by placeholders. For example, the user might say `if` and get `if (null predicate) then { null statement }`.

Another form of code entry is to support textual macro expansions. Current users of voice recognition utilities rely on extensive macro support provided by the tools to enter text. While we can better implement many uses of these macros with template instantiation and better understanding of the words that they speak, we must continue to support this kind of legacy tool.

Finally, using the results of our proposed user studies, we will explore new ways to enable more natural expression of code. For example, we wish to enable a programmer to say “repeat ten times” to get a for loop in Java, or “create an integer named foo” to declare a new variable. If a company enforces coding style rules, these could be incorporated into the framework. For example, if Hungarian notation were required, when the user said “create an integer named foo”, we would insert the code `int iFoo` to conform to the proper style. We feel that while at first it may seem that the programmer may be saying more than they would have to if dictating code directly, the shift towards higher-level conceptual programming will far outweigh the extra verbiage.

### 6.3 Program Navigation and Browsing

Software developers that use text editors as their interface tend to use text-based navigation, moving to new characters, lines, buffers, or files. They use text-based search for character strings in character buffers. The maintenance by an interactive development environment of structural, semantically annotated, program views enables the user to navigate using structural and semantic targets (e.g. the next statement, the outer declaration of `I`, the use of `J` as a parameter to `foo`) and to search for patterns having certain semantics. In earlier systems, such navigation tends to be menu or function-key driven, with a fixed set of syntactic and semantic constructs, and a fixed set of actions [55].

As part of the Harmonia project, researchers will develop methods for constructing patterns and for searching for them in programs. As part of this dissertation, we will explore using the techniques used for spoken authoring to enable the user to verbalize the construction of search patterns. Our analysis technology will translate the verbal search patterns into lower-level queries which will be run by the Harmonia framework.

A salient feature of a powerful navigation and modification facility is the ability to refer not only to commands that indicate what operation to perform, but also to constructs found in the programming language and in the program. In this manner, the user might be able to say “go to method Foo in class Bar” to move the cursor to the beginning of the `Bar.Foo` method. Likewise, we can exploit the language grammar to constrain program modifications. If the user says “replace the second if statement’s predicate by x is greater than seven”, the phrase “if statement’s predicate” indicates the predicate nonterminal of the grammar rule for an `if` statement, “the second” says to construct a tree pattern in which there is an `if` statement followed by a second `if` statement, and “x is greater than seven” is a boolean expression with which to replace the existing predicate’s leaf text terminals. We will develop the necessary technology to automate the construction of a grammar for referring to language constructs (pieces of the language dictation grammar) in a navigation context.

### 6.4 Program Editing

Program navigation allows the programmer to position the cursor at various parts of the program. The user must also be given the ability to manipulate what he finds. In addition to the traditional text editor
functionality of deletion and insertion of characters and blocks of text via the keyboard and mouse, the Harmonia framework will enable the user to perform insertion, deletion and replacement operations on structural and semantic constructs. If the search pattern language is sufficiently rich, it appears that a fixed set of commands could be added to derive the necessary functionality. It will be part of this dissertation to make it possible to recognize those commands in the spoken language.

As part of the Harmonia project, we will be developing the language analysis infrastructure to support code transformations. Transformations can often be the most efficient form of software maintenance. For instance, a user may want to factor several variables into its own data structure. This has effects on all pieces of code that reference those variables. If the transformation is performed, it must preserve program semantics. Part of this dissertation will explore ways to verbalize code transformations. Since much of this process involves speaking about multiple code entities, multi-modal interaction between voice and mouse pointing has the potential to make it relatively simple to perform.

6.5 On-screen Representation

If we modify an existing programming language’s syntax to be more easily spoken than the textual one, what does the user edit on the screen? If we only display the traditional, textual representation, then we lose input transparency touted by many in the HCI community. On the other hand, while the modified syntax may be easier to speak, it may not be very easy to read.

In the spirit of the Ensemble project, we propose to build an editor that displays multiple representations of the code being edited. Each representation can be tailored for the input mode, thus it would be displayed in the grammar appropriate for that input mode. In our editor which supports voice and keyboard entry, we can show two screens – one has the direct input from the user (with formatting services provided by Harmonia), the other shows what the code looks like in the traditional syntax.

Since our modification of the programming language syntax does not change its abstract syntax, but merely its concrete syntax, we will be able to automatically map edits made in one buffer into changes for the other buffers. We can do this by mapping all changes through the abstract syntax tree kept by Harmonia. This technology will build on prior work done during the Pan project [10].

6.6 Prototype Editors

We plan to build several prototype editors to test our algorithm and technology development. Our zeroth prototype (so named because it should only take a week or so to build, and will work without any new technology) will hook the ViaVoice speech recognizer up to XEmacs (running under Linux). Harmonia will also be attached to this XEmacs, but no advanced features will be used. The primary mode of input will be keyboard-based, but we will build a command grammar for ViaVoice to insert predefined grammar template instantiations into the buffer. We will design one of these templates for each major portion of the Java grammar and hard-code each one into ViaVoice (eventually we will use Harmonia’s ability to read and manipulate, at run-time, the programming language grammar in use for the current document to instantiate a grammar production). The grammar used will be the standard Java textual syntax. Once the templates have been instantiated, the rest of the user input will be the traditional one by keyboard and mouse. The purpose of this prototype is to explore the simplest form of voice recognition-enhanced programming and compare it to keyboard-based programming.

Our first prototype will add real template instantiation support, making command keywords from ViaVoice instantiate grammar productions from the current language grammar. In addition, we will annotate the grammar with the command keywords and autogenerate the ViaVoice command grammar file. We also plan to add a programming language grammar-aware navigation and editing command and control grammar. This requires the ability to explore Harmonia’s parse tree for the given document, as well as query semantic-based services which will provided by Harmonia.

Later prototypes will add pieces of technology as they become available. We will enable the Spoken Java grammar and allow the user to code in the more natural style afforded by it. We will add user-visible feedback mechanisms to enable ambiguity resolution (manual parse tree disambiguation) and improve voice recognition accuracy (through the use of the tools provided by the speech recognition system). As we develop
our automatic disambiguation techniques, we will add these as well, in order to spare the user a lot of needless
dialogue from the editor about disambiguating parse trees.

Towards the end, we may experiment with multi-modal interaction to integrate gestural input (from a
mouse or pen) with spoken language input. This would enable, for instance, a user to invoke an operation
on an “unnamed” language construct by pointing to it and speaking a command.

We received another suggestion that the manual disambiguation dialogue could take place verbally,
utilizing text-to-speech engines to ask the user which parse is correct. In addition to being somewhat
intriguing, 6 this opens up a new area of research, program text-to-speech. How might a Java computer
program be verbalized? If it is translated first to Spoken Java and then verbalized, but if the user programmed
using higher-level program composition, you would definitely not want to lose that information when speaking
the code. This work could lead to a very interesting Master’s project for some new graduate student.

7 Cognitive Models

With this section, we switch gears a bit and move back from the technical issues related to voice-based
programming towards the cognitive issues. How is code spoken? How is this different from writing code?
What effect might speaking code have on our problem-solving ability? How do novices, who have an in-
complete understanding of computer programming, talk about code? How is this different than an expert
talking about code? We hope to gain insight on these questions and contribute to the body of research on
expressing oneself through code.

Many studies investigating how people learn to program are concerned with how students understand
code [44, 22, 45, 19] and how to enhance that process [34, 11, 13]. Their analysis of students’ mental activities
takes many forms, but one effective idea is to use a “think aloud” protocol, where the students are given
a task and asked to verbalize their thoughts while working through the task. The students’ behaviors are
recorded and correlated with their verbalized thoughts to help researchers gain insight into the problem-
solving process. We thought that “think aloud” protocols might give us ideas on how people verbalize their
coding process, but they break down when the problem-solving process itself is verbal. In addition, while
they expose the programmer’s thought process, it isn’t a deliberate communication, as it would be when the
programmer is intentionally expressing their conceptual and pragmatic ideas about programming to others.

7.1 Writing vs. Speaking

Programming is a form of communication which usually takes place in a written medium. Even though
the code itself is easily written down, programmers need to convince others of their ideas through program
documentation, verbal explanations, gestures, and drawings. While these extra forms of communication
could be considered irrelevant to the software artifact being created, they are an important part of the
software development process. Very few programmers work alone, and very few work without someone
evaluating their work. A programmer’s ability to communicate is vital for proper appreciation of their work.

What is the difference between the written output of a programmer and his verbal output? The way that
people express themselves when speaking is fairly different than when writing. Three of these differences
(there are no doubt more) are relevant to programming. First, we convey a lot of meaning through our
vocal expression. Whereas writing uses vocabulary, punctuation and capitalization, speaking uses loudness,
timbre, pitch, pauses and the vernacular. Current speech recognition tools can not pick up any of these
qualities other than the words themselves, which leads to less effective communication with the computer
than by writing.

Abstraction is very important to spoken language communication. When someone explains a concept to
you, they first explain it at a high-level. If at first you don’t understand, they drop down to a level of greater
detail and explain how that works. Contrast this process to writing programs. The high-level architectural
system structure is kept in the programmer’s head, while instead, all the details are written down to be
understood by the computer.

6One of the author’s first projects enhanced the Eliza doctor program in XEmacs to accept all I/O via speech; this made
Eliza a whole lot more therapeutic than when it was keyboard-based.
Lastly, when speaking, consistency and correctness are inferred by context. People repeat words, drop words, misuse words, flub pronunciation, and employ their stream of consciousness. When writing code, programmers revise and edit their work until it is perfect. When the code is sent to the computer, it must be completely correct in order for the computer to understand it.

These three differences: expressiveness, abstraction and correctness, are key to understanding how the programming and software development process will differ when moving to programming by voice from programming by keyboard. Current commercial speech tools do not support recognition of vocal expression. In order to support the abstraction that people bring to their spoken words, any program by voice tool must eventually support higher-level linguistic descriptions of programming. As we stated above, anecdotal evidence suggests that when users talk about programs to one another, they speak in terms of constructs and semantic properties rather than textual entities. And finally, speech filtering and error correction cannot solely occur in the speech recognizer; in order to properly fix the error, filtering and error correction must take place in the analysis layer that can handle it best, either in the lexer, parser or semantic analyzer.

8 Evaluation of Program by Voice Systems

We will explore the cognitive, linguistic and practical issues related to spoken software development in several user studies. First, we will explore how students (and expert programmers) communicate verbally with each other to solve software engineering tasks. Then, we will perform several Wizard of Oz studies to explore designs for verbal program dictation, navigation and editing. These will be followed by experiments on our actual editing framework as it is being built.

Our first studies will take place at the beginning of our project. We want to see how two students communicate with each other in several programming tasks: program composition, program navigation and program editing. The first study will have one student dictate verbally pre-written code to another student. The second study will see how two students solve a simple programming task, where one of them works on the programming task and verbally indicates to the other student what to type into the computer. The second student will not work on the problem. We will be watching the students to see what things they say to each other to communicate the code, and how they go about solving the problem. As we’ve stated previously, programmers do not code from top to bottom; instead, they jump around. We want to identify how this behavior manifests itself.

We will record these sessions using an addition to harmonia-mode for XEmacs, which will use the speech recognition interface to record audio clips (and text translation) of the words spoken by the first student. The editor will record all activities into a log. This will allow us to replay the session again without having to reconduct the experiment. It will also allow us to revisit the original sessions to mine more interesting ways to program as we develop new technology for this project.

Our Wizard of Oz experiments will take the form of a student performing a task verbally using prototypes of the Spoken Java grammar. I will interpret their words using the same grammar while typing the results of their actions into a text editor. We will experiment with students dictating pre-written code, editing pre-written code to alter its function, and solving a problem (of the difficulty of a problem in CS61b) from scratch.

The initial set of experiments are intended to inform our editor and grammar design process. Later, we will repeat these experiments (as well as perform experiments on our prototype editor) and compare the students’ performance relative to traditional programming techniques using the keyboard and mouse. Additionally, once students experiment on the actual prototype editor, we will also compare performance using voice-only techniques with voice-plus-keyboard-plus-mouse.

8.1 Evaluation Metrics

What metrics are relevant to evaluation of spoken software development? Speed to completion of task can be relevant, but only for the actual code entry task. Do programmers fall back to direct program dictation, or do they take advantage of the high-level linguistic forms of input?

We also would like to look at the learning process. How hard is it to learn to program by voice? How is this different for novices than experts? How often does the programmer lose track of where they are? Why?
Is it due to improper feedback from the editor, or is it due to inaccurate speech recognition? How many and what type of mistakes does the programmer make? How often does he not know what to say or not know how to verbalize something? This includes not knowing what to call a particular language construct (before we support pointing gestures), and also includes misuse of the programming language grammar. Does the user experience voice strain? How much training in speech tools is necessary before the proper techniques to avoid RSI are learned? Finally, do people like programming by voice? What about it appeals to them? What tasks are annoying or difficult to accomplish?

9 Dissertation Contributions

This dissertation will make several important contributions: enhancing the Harmonia analysis framework, exploring how programmers verbalize the software development process, developing high-level verbal software development languages to naturally support program dictation, composition, navigation, browsing, and editing, inventing and evaluating the Spoken Java dialect, and conducting user studies to evaluate the impact of voice on actual software development.

By the time we are finished, we will have enhanced three key components of the Harmonia analysis framework: the lexer, the parser, and the semantic analyzer. In the lexer, we will support alternate lexeme values for a given lexeme. We will be able to pass multiple alternate lexemes to the parser for a given input word. We will also perform rule-based combinatoric concatenation of the lexical token stream. In the parser, we will support alternate lexemes by performing multiple shifts on each relevant parse graph. We will also enable each parser to have its own pointer into the token stream (with concatenation information as annotations) preserving GLR’s parser merging property. Additionally, we will better support speech input by enhancing the parser with the ability to skip tokens from the input to recover parsing when the user speaks extraneous words not found in the grammar. Finally, we will implement a set-based, incremental, demand-drive attribute grammar evaluator that works on ambiguous parse forests. In addition, this evaluator will be able to perform automatic disambiguation of parse forests (by pruning incorrect parse trees).

Second, we will conduct user studies to see how both novices and experts verbalize the software development process in four different task areas: program dictation, composition, navigation and browsing, and editing. These user studies will inform our designs for spoken forms of language grammars that support the five activities.

Third, we will explore designs for spoken software development: program dictation, composition, navigation and browsing, and editing. We will develop methods to integrate speech recognition with program composition to make the coding process simpler, faster and easier. We will also identify new voice services required for editing, and error correction. We will support manual disambiguation dialogues with the user to resolve irresolvable ambiguities discovered by the language analysis process.

Fourth, we will develop, refine and evaluate the Spoken Java dialect of the Java programming language. This dialect will be designed with the explicit purpose of making Java easy to speak and will be the main testbed language for this dissertation.

Finally, we will conduct user evaluation studies to explore the cognitive, linguistic and practical aspects of the spoken software development environment that we have created.

10 Timeline

- Spring 2001:
  - Build 0th prototype of program by voice editor. This will use speech only for template expansion keywords. The rest of the input will be via keyboard and mouse.
  - Conduct user studies to see how two students interact in software development tasks.
  - Develop logging capability for harmonia-mode to integrate voice and text edit logging in XEmacs.
  - From the results of the user tests, design Spoken Java dictation grammar, program composition, navigation and editing grammars.
– Run Wizard of Oz experiments to try out the first Spoken Java grammar and editing/navigation grammars.

- **Summer 2001:**
  – Begin adding lexer enhancements to support ambiguities arising from voice input.
  – Modify parser to accept lexer ambiguities.
  – Add manual disambiguation interface to the parser and harmonia-mode in XEmacs.
  – Revise Spoken Java grammar (and editing/navigation modes) based on prior experiments.

- **Fall 2001:**
  – Build incremental attribute grammar evaluator
  – Identify attributes necessary for automatic disambiguation of parse forests.
  – Run more user experiments (both Wizard of Oz and actual use of prototype editor) with updated Spoken Java and editing/navigation grammars.

- **Spring 2002:**
  – Develop feedback algorithms to improve recognition accuracy through online tracking of the lexing, parsing and semantic analysis.
  – Develop training methods to teach students how to program by voice.
  – Conduct usability experiments on Spoken Java grammar and editing tool (with support for automatic disambiguation).

- **Fall 2002 – Spring 2003:**
  – Finish up implementation and evaluate user studies.
  – **Write dissertation and graduate!**

### 11 Future Work and Conclusion

In the future beyond the scope of this dissertation, we see the development of course additions to classes like CS61b (Introduction to Data Structures) and CS164 (Introduction to Compiler Design), both taught in Java, to support programming by voice activities. We envision additional user studies to investigate the impact of verbal programming on novice programmers, as well as explore how experts, both with RSI and without, adapt to this new technology. Moving to a classroom setting brings new challenges, such as identifying methods to support collaborative verbal software development in noisy lab sessions, and understanding how best to encourage students’ expression in software development tasks.

We also see new designs for verbalizing other programming languages, such as C, C++, Javascript, and COOL (the language used in CS164). In addition, since we speculate that native English speakers and non-native English speakers verbalize code differently, we would like to see grammars for C, Java and the like designed for non-native English speakers. How might these dialects be different than those designed for native speakers? Finally, we envision support for gestures and drawings as alternate means of program expression. Perhaps the user could a draw control flow or data flow diagram as part of their code. Correctly and accurately parsing drawn pictures is a tough research problem and will require more research before the state-of-the-art becomes mature enough to tackle this problem.

In conclusion, the work described in this dissertation proposal will help to answer three important questions. One, how do programmers verbalize the software development process? How is this different than writing? Two, can we make a computer understand speech in support for the software development process? What algorithms and techniques will be necessary to solve this problem? And three, can we create...
a software development environment that supports this kind of communication in a feasible manner? Will this environment make programming easier and more efficient?

The current state-of-the-art in programming by voice leaves much to be desired, both in terms of ease of use, and the quality of recognition. By applying our expertise in programming language analysis to the output of a speech recognizer, we hope to create an environment that will support a more natural mode of verbal expression by programmers. This framework will also support high-level linguistic composition, editing, and navigation activities beyond simple program dictation to create a complete solution for the programmer.

References


Palm, Inc. *Palm Pilot (tm)*.


