Spoken Language Support for Software Development

Andrew Begel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-8
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-8.html

January 26, 2006

Acknowledgement

The work conducted in this dissertation has been supported in part by NSF Grants CCR-9988531, CCR-0098314, and ACI-9619020, by an IBM Eclipse Innovation Grant, and by equipment donations from Sun Microsystems.

Spoken Language Support for Software Development

by

Andrew Brian Begel

Bachelor of Science (Massachusetts Institute of Technology) 1996
Master of Engineering (Massachusetts Institute of Technology) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Susan L. Graham, Chair
Senior Lecturer Michael Clancy
Professor Marcia Linn

Fall 2005

The dissertation of Andrew Brian Begel is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2005

Spoken Language Support for Software Development

Abstract

Spoken Language Support for Software Development

by

Andrew Brian Begel

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Susan L. Graham, Chair

Programmers who suffer from repetitive stress injuries find it difficult to program by typing. Speech interfaces can reduce the amount of typing, but existing programming-by-voice techniques make it awkward for programmers to enter and edit program text. We used a human-centric approach to address these problems. We first studied how programmers verbalize code, and found that spoken programs contain lexical, syntactic and semantic ambiguities that do not appear in written programs. Using the results from this study, we designed Spoken Java, a semantically identical variant of Java that is easier to speak. Inspired by a study of how voice recognition users navigate through documents, we developed a novel program navigation technique that can quickly take a software developer to a desired program position.

Spoken Java is analyzed by extending a conventional Java programming language analysis engine written in our Harmonia program analysis framework. Our new XGLR parsing framework extends GLR parsing to process the input stream ambiguities that arise from spoken programs (and from embedded languages). XGLR parses Spoken Java utterances into their many possible interpretations. To semantically analyze these interpretations and discover which ones are legal, we implemented and extended the Inheritance Graph, a semantic analysis formalism which supports constant-time access to type and use-definition information for all names defined in a program. The legal interpretations are the ones most likely to be correct, and can be presented to the programmer for confirmation.

We built an Eclipse IDE plugin called SPEED (for SPEech EDitor) to support the combination of Spoken Java, an associated command language, and a structure-based editing model called Shorthand. Our evaluation of this software with expert Java developers showed that most developers

had little trouble learning to use the system, but found it slower than typing.

Although programming-by-voice is still in its infancy, it has already proved to be a viable alternative to typing for those who rely on voice recognition to use a computer. In addition, by providing an alternative means of programming a computer, we can learn more about how programmers communicate about code.

_____

Professor Susan L. Graham
Dissertation Committee Chair

To Noah and Robbie,

who encouraged me to take the relaxed approach to graduate school.

And to Sean, who said to hurry up and finish already!

Contents

List of Figures

List of Tables

Acknowledgments

I cannot begin to acknowledge anyone without first recognizing the role that Susan L. Graham, my advisor, played in my graduate career. Sue has this amazing ability to listen to a young, idealistic graduate student ramble about his interests in computer science and somehow cause him to think that everything he wants to do not only has coherence and purpose, but that it is exactly the kind of work that Sue wants to do. Of course, the only option is to hook up with her and start doing some great science. When she is available to talk, she is always willing to give you as much time as you want, without even glancing at the clock. Sue, thank you for all the advice you have given me over the years, and for your willingness to let me work on such a massive dissertation.

I would like to thank the other members of my dissertation committee, Michael Clancy and Marcia C. Linn, for their encouragement and enthusiasm. I have never seen a dissertation topic cause so much talk and excitement among professors, both about whether it could succeed at all, and about the kinds of effects it might have on people if it did. Thank you too to James Landay for sitting on my qualifying exam committee and ensuring that I considered the people who would use my technology as highly as the technology itself.

Jennifer Mankoff taught me how important it is to design computer tools for people with disabilities. This concern dovetailed with my research at a perfect time to wield great influence on the work.

I would like to thank my long-time Harmonia project officemates: Marat, Johnathon and Carol. Marat, you have been my counterpart through the long years of graduate school. You helped me immensely at the beginning when I needed it; I learned so much from you. In the middle years, we both emerged as leaders in our own right, and at the end, you are still right there with me, developing ideas that dovetail with many of my own. Oh, and by the way, I win. Johnathon and Carol, you have both provided a congenial atmosphere to the office and taught me much about real Californians. You have also eagerly listened to all my advice. I cannot say that about too many other people. To Caroline Tice and David Bacon, Sue's former students who overlapped with me: I did not understand what you meant at the time, but the wisdom has finally come. Thank you.

Thank you to all the Berkeley undergraduates who did research with me over the years. You guys did a lot of coding in a selfless manner, and I appreciate every bit of it. To name names: John J. Jordan, Stan Sprogis, Michael Toomim, John Firebaugh, Tse-wen Tom Wang, Tim Lee, Jeremy Schiff, Dmitriy Ayrapetov, Erwin Vedar, Brian Chin, Duy Lam, Stephen McCamant, John Nguyen, and Alan Shieh. I especially want to thank John J. Jordan for designing and implementing

Shorthand and Michael Toomim for designing and implementing Harmonia-Mode for XEmacs and Linked Editing. You two have immense potential and have become good friends; I am proud to have known you.

I would like to thank all of the participants of the spoken programs, SpeedNav, and SPEED user studies. Your participation helped make this dissertation a success. I want to thank my colleague Zafrir Kariv, for helping develop the ideas and tools in SpeedNav, and helping run and analyze that data from the subsequent user study. We would like to thank all the students in the Assistive Technologies class in which this research was conducted. In addition, Professor Jennifer Mankoff gave us valuable advice and feedback on SpeedNav. We would like to thank Michael Toomim for helping make many of the changes necessary to render and edit voice comments for the commenting by voice project. We also thank Marat Boshernitsan for his assistance in modifying the Harmonia program analyses to accept voice comments without crashing.

For my entire undergraduate years and during most of my graduate career, I was a stealth graduate student at MIT working on the StarLogo project with Mitch Resnick, Brian Silverman and Eric Klopfer. I have played many roles in this work, from coder to designer, to leader and advisor. I feel proud to call you all my mentors and colleagues.

Throughout my years at Berkeley, I have met some amazing people who I will never forget, nor leave behind. Dan Garcia, you are a teaching powerhouse, a great mentor and a wonderful colleague. Tao Ye, Helen Wang, Adam Costello, Tom Zambito, Yatin Chawathe, Steve and Munazza Fink, Dan Rice, Ben Zhao, Steve Czerwinski, Jason Hong, Jimmy Lin, Rich Vuduc, Chennee Chuah, Mark Spiller, Tina Wong, Josh MacDonald, and Michael Shilman: you all let me wander into your offices to gab, discuss tough research questions, and ask really hard compiler questions. You did great on the first two, but seriously, I had to solve almost all the compiler problems myself.

Many MIT friends joined me out here in the Bay Area and helped me escape from the ivory tower to have fun once in a while. Thanks to Zemer and Coleen, Andy and Sarah, Nick and Diane, Eytan and Sara, Michelle and Robert, Rachel and Paul, Sasha and Sonya, and Jordan.

Finally, I cannot give enough thanks to my partner, Sean Newman. Your love, compassion, encouragement and companionship have sustained me through the past one and half years. A requirement to finish every Ph.D. is a person who gives you the reason to finish up and leave. Sean, you are my person. Thank you.

The work conducted in this dissertation has been supported in part by NSF Grants CCR-9988531, CCR-0098314, and ACI-9619020, by an IBM Eclipse Innovation Grant, and by equipment donations from Sun Microsystems.

Chapter 1

Introduction

Software development environments can create frustrating barriers for the growing numbers of developers who suffer from repetitive strain injuries and related disabilities that make typing difficult or impossible. Our research helps to lower these barriers by enabling developers to use speech to reduce their dependence on typing. Speech interfaces may help to reduce the onset of RSI among computer users, and at the same time increase access for those already having motor disabilities. In addition, they may provide insight into better forms of high-level interaction.

This dissertation explores two questions:

1. Can software developers program using speech?
2. How can we make a computer understand what the developer says?

Our thesis is that the answers to these questions are 1) yes, programmers can learn to program using their voices, and 2) by creating a compiler-based speech understanding system, a computer can successfully interpret what the programmer speaks and render it as code in a program editor.

To explore this thesis, we took a three-pronged human-centric approach. First, we studied developers to learn how programmers might naturally speak code. After all, programming languages were designed as written languages, not spoken ones; while intuition says that programmers know how to speak code out loud, what they actually say and how they actually say it has never before been formally studied. Second, we used the results of the study to design a spoken input form that balances the programmer's desire to speak what feels natural with the ability of our system to understand it. Our contribution is to use program analyses to interpret speech as code. By

extending these analyses to support the kinds of input that speech engenders, we can adapt existing algorithms and tools to the task of understanding spoken programs. The main artifact of our work is realized as a software development system that can understand spoken program dictation, composition, navigation and browsing, and editing. Third, we evaluated our techniques by studying developers using our programming-by-voice software development environment to create and edit programs. We found that programmers were able to learn to program by voice quickly, but felt that the use of speech recognition for software development was slower than typing. In addition, programmers preferred describing code to speaking it literally. The programmers all felt that they could use voice-based programming to complete the programming tasks required by their jobs if they could not type.

We identify four major challenges in this work:

1. **Speech is inherently ambiguous** – People use homophones (words that sound alike, but have different spellings and meanings), use inconsistent prosody (pitch and modulation of the voice) confusing others as to whether they are asking a question or making a statement, flub their words or word order as they speak, or use stop words, such as "uh," "um," and "like" along with other speech disfluencies. Usually, we understand the other person because we can bring an enormous amount of context to bear in order to filter out all the inappropriate interpretations before they even rise to the level of conscious thought. Even with all this context, humans sometimes have trouble understanding one another. Imagine a computer trying to do this!

2. **Programming languages and tools were designed to be unambiguous** – The entire history of programming languages is filled with examples of programming languages that were designed to be easy to be read and written by machines rather than by humans. All of the punctuation and precision in programs is there in order to make program analyses, compilers and runtime systems efficient to execute and feasible to implement. This sets up an unfortunate situation for voice-based programming due to the clash between the inherent ambiguity of speech and the lack of preparation for ambiguity found in programming languages and tools.

3. **Speech tools are poorly suited for programming tasks** – Speech recognizers are designed to transcribe speech to text for the purpose of creating and editing text documents. They are trained to understand specific natural languages and support word processing tasks. Pro-

gramming languages are similar to but not the same as natural languages in ways that make recognition difficult. In addition, the kinds of tasks that programmers undertake to create, edit and navigate through programs are very different than word processing tasks. Commercial speech recognizers do not come with any tools designed for programming tasks.

4. **Programmers are not used to verbal software development** – Programmers have been programming text-based languages with a keyboard for many decades. While they may talk about code with one another, they do not speak code to the computer. There will be a learning curve associated with programming-by-voice, just as there is a learning curve associated with speaking natural language documents. Certain dictation techniques are better matched with computer-based voice recognition than others, and learning what those techniques are for programming will need to be one of the objects of study.

In the remainder of this chapter, we describe in more detail the problem of programming using voice recognition, describe related work in the field, and sketch out our solution in the form of new programming languages and environments, and new lexing, parsing and semantic analysis algorithms to make them work.

1.1 Speech Recognition

Before the advent of the integrated development environment (IDE), software developers used text editors to create, edit, and browse through their programs. IDEs improve usability via a graphical user interface and better tools, but the main work area is still a text editor. Programmers with RSI and other motor disabilities can find these environments difficult or impossible to use due to their emphasis on typing. According to the United States Department of Labor, Bureau of Labor Statistics (BLS) [70], in 1980, 18% of all workplace injuries were due to RSI. In 1998, the number had risen to 66%! This is the largest category of worker-related injury in the USA, and causes the longest work outages (median 32 days for carpal tunnel syndrome, 11 days for tendonitis in 2003). In 2004, according to the BLS, there were around three million professional computer programmers. BLS surveys since 1992 (see Table 1.1) show that hundreds of them every year suffer from repetitive motion injuries that cause them to lose working days. Assuming that some incidences of RSI are not reported, there could easily be thousands of software developers that have trouble staying productive in their work environment due to RSI.

Year	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003
Cases	450	631	538	777	386	432	328	699	632	725	742	270

Table 1.1: Number of repetitive motion injuries per year for computer and data processing services jobs involving days away from work. From the U.S. Department of Labor, Bureau of Labor Statistics.

We believe that speech interfaces could be employed to reduce programmer dependence on typing and help offset the chances of developing RSI, as well as helping those who already suffer from RSI. Speech recognition has been commercially available for desktop computers since the mid 1990s. It was only then that commodity hardware had the horsepower and memory required to run the huge hidden Markov models that power these recognition engines. Several speech recognizers have enjoyed commercial success: Nuance's Dragon NaturallySpeaking [69], IBM's ViaVoice [39], and Microsoft's Speech SDK [38].

The primary use of desktop speech recognition is to create and modify text documents. Recognizers support two modes of operation: dictation and command. Dictation transcribes what the user speaks, word for word, and inserts the results into a word processor such as Microsoft Word. Commands are used afterwards to format the prose, for example, to add capitalization, fix misspellings, and add bold, italics and other font styles. In addition, commands are used to navigate through the windows, menus, dialog boxes and the text itself.

Speech recognition is not perfect. Recognition accuracy, while usually in the 99% range with adequate software training, requires that speakers continuously scan speech-recognized documents for typos and misrecognized words. Accuracy suffers when users have accented speech, speech impediments, or inconsistent prosody (such as if the user has a cold or is hoarse). Noisy environments degrade accuracy as well. Misrecognition in command mode can lead to frustration trying to manipulate a graphical user interface that was never designed for vocal interaction. The physical motions involved in typing and mousing are much quicker than their verbalization, and are much less likely to be misinterpreted by the computer. Numerous studies show problems with even basic usage of speech recognition, including errors (due to both the recognizer and the user, especially during misdictation correction) [47], limited human working memory capacity for speech [49], and limited human ability to speak sentences that conform to a particular grammar [87]. In addition, users must re-learn basic word processing techniques using voice rather than keyboard and mouse. These problems and the steep initial learning curve cause, for most users, speech recognition-based editing to be significantly (and often prohibitively) slower than editing using keyboard and mouse.

1.2 Programming-By-Voice

Desktop speech recognition tools were designed for manipulating natural language text documents. This makes them poorly suited for programming tasks because they are based on statistical models of the English language; when they receive code as input, they turn it into the closest approximation to English that they can.

Efforts to apply speech-to-text conversion for programming tasks using conventional natural language processing tools have had limited success. IBM ViaVoice and Nuance Dragon NaturallySpeaking support natural language dictation and commands for controlling the operating system GUI, application menus, and dialog boxes. Inside a text editor, the most common place to find program code, supported editing commands are oriented towards word processing, supporting font and style changes and clipboard access. Commands for common programming operations, such as structurally manipulating text, are absent.

Some disabled programmers have successfully adapted the command grammars that drive speech recognition for programming tasks. However, these grammars are necessarily prescriptive and provide only limited flexibility for ways of programming not anticipated by the authors. Command grammars are not context-free; they cannot have recursion. This immediately limits an author's ability to describe programming languages with these grammars. In addition, grammars are not easily extensible by end-user programmers, unless they spend a great deal of time analyzing their own programming behaviors and using this information to create their own command grammars.

Jeff Gray at University of Alabama speech-enabled the Eclipse programming environment [90]. The word "speech-enabled" means that a command grammar was created for the menus, dialog boxes and GUI of the IDE. The program editor itself was not accessible by speech. T.V. Raman has speech-enabled Emacs, making accessible all of the Meta-X commands and E-Lisp functions defined within [80]. In addition to speech input, Raman has also enabled Emacs for speech output. Even without a screen reader, Emacs can now output its text and commands in spoken form.

Speech-enabling IDEs is only the first step to making a usable programming environment. To author, edit and navigate through code by voice, developers need to speak fragments of program text interspersed with navigation, editing, and transformation commands. Recent efforts to adapt voice recognition tools for code dictation have seen limited success. Command mode solutions, such as VoiceCode [20, 101], sometimes suffer from awkward, over-stylized code entry, and an inability to exploit program structure and semantics. VoiceCode compensates for this lack of analysis by providing detailed support for a large number of language constructs found in Python and C++

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

(a)

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

(b)

Figure 1.1: To get the for loop in (a), a VoiceCode user speaks the commands found in (b).

code. An example using VoiceCode to enter a for loop is shown in Figure 1.1. The commands are interpreted as follows.

1. **for loop**: Inserts a for loop code template with slots for the initializer, predicate and incrementer.
2. **after left paren/semi/left brace**: Command to move to the next slot in the code template. Analogous commands exist to move to the previous slot. Once all slots have been filled in, future navigation is based on character distance and regular expression searches.
3. **declare india**: Creates a new variable named "i." Most speech recognizers require the speaker to use the military alphabet when spelling words.
4. **of type integer**: A command modifier to "declare", that adds the type signature to a declaration.
5. **assign zero**: Assignment in VoiceCode is "assign," not "equals."
6. **recall one**: Identifiers in VoiceCode can be stored in a cache pad, a table of slots each of which is addressable by a number from one to ten. To reference a previously verbalized identifier, the user says "recall" and the number of the slot.
7. **increment**: VoiceCode's way to say "plus plus."

Lindsey Snell created a program editor that eased some of the awkwardness of the command grammar approach by automatically expanding keywords into code templates [92]. In addition, her work could use temporal lexical context to detect when the user was trying to say an identifier rather than a keyword, and act appropriately. For example, when multiple words are spoken

in a row, they might all form part of the same identifier. Context-sensitivity enables the environment to detect this and automatically concatenate the words together in the appropriate style for that programming language. This context is limited to the initial dictation of the code, however. The system does not appear to take advantage of spatial context to provide the same support for editing pre-existing code. In addition, the context is detected lexically, which limits the ability to provide appropriate behavior inside function bodies where the necessary context is often ambiguous.

Taking a different approach, the NaturalJava system [79, 78] uses a specially developed natural language input component and information extraction techniques to recognize Java constructs and commands. This is a form of meta-coding, where the user describes the program he or she wishes to write instead of saying the code directly. Parts of that work are promising, although at present there are restrictions on the form of the input and the decision tree/case frame mechanism used to determine system actions is somewhat ad hoc. Worse, the tool is not interactive, but rather a batch processor that produces code only after the programmer has described the entire section of code.

Arnold, Mark, and Goldthwaite [4] proposed to build a programming-by-voice system based on syntax-directed editing, but their approach is no longer being pursued.

An important part of programming is entering mathematical expressions. Fateman has developed techniques for entering complex mathematical expressions by voice [29] that could be used in our programming-by-voice solution.

Voice synthesis has been applied to speaking programs. Francioni and Smith [32, 91] developed a tool for speaking Java code out loud for blind programmers. Punctuation is verbalized in English, and structure beginnings and ends are explicitly noted (with associated class and method names when applicable). Modulation of speech prosody is used to indicate spacing, comments and special tokens or structures. Verbalization of programs like this can be used in a programming-by-voice system to train developers to speak Java code out loud.

1.3 Our Solution

Our research adapts voice recognition to the software development process, both to mitigate RSI and to provide insight into natural forms of high-level interaction. Our main contributions are to use a human-centric approach in the design of our voice-based programming environment, and to use program analysis to interpret speech as code. This enables the creation of a program editor that supports programming-by-voice in a natural way.

We began our work by studying programmers to learn how they might speak code in a programming situation without any advanced training or preparation. We wanted some ground truth data on which to base a new input form for voice-based programming, one that would be both natural to speak and easy to learn. We found that there are some significant differences between written and spoken code, categorizable into roughly four classes: the lexical, syntactic, semantic, and prosodic properties of input. There is considerable lexical ambiguity, since spoken text does not include spelling, capital letters or an indication of where the spaces in between the words belong. Syntactically, the punctuation that helps a compiler analyze written programs is often unverbalized, leading to structural ambiguities. In addition, some phrases from the Java language prove difficult to speak out loud due to differences in sentence structure from English. Semantically, programmers speak more than the literal code; they paraphrase it, and talk *about* the code they want to write. Finally, we found that prosody is often used by native English speakers to disambiguate similar sounding phrases, but is not employed by non-native speakers.

We then studied voice recognition users to learn how existing speech recognition tools supported other important tasks, such as navigation through a document. We found that navigation commands provided by speech recognizers fall into a variety of categories, ranging from context-independent commands to navigate using relative and absolute coordinates, to context-dependent commands to navigate using absolute coordinates, and to search tools like the Find dialog in a word processor. Each of these mechanisms suffers from two problems: too much delay between the start of the navigation and the end because too many spoken commands are required, and a high cognitive load, due to the need for constant supervision and feedback of the navigation process and the correction of recognition errors.

We attempted to design an alternate form of navigation to address these issues, but ran into technology problems caused by long voice recognition delay. In the process, however, we learned several lessons about navigation that we have applied to a new technique that is especially appropriate for navigation through programs: context-sensitive mouse grid. Context-sensitive mouse grid enables programmers to identify and select program constructs by number in a hierarchical manner. It works well because the most successful adaptations of user interfaces for speech recognition rely on labeling possible user selections with on-screen numbers in order to allow the user to simply speak the numbers that he sees. Labeling interesting locations or actions on-screen means that a user does not have to verbalize the locations or actions himself, decreasing the cognitive load and increasing the recognition success rate.

Based on our studies of programmers, we have created Spoken Java, a variant of Java

which is easier to verbalize than its traditional typewritten form, and an associated spoken command language to manipulate code. Spoken Java more closely matches the verbalization in our study than does the original Java language. In this program verbalization, programmers speak the natural language words of the program, but must also include verbalizations of some punctuation symbols. Spoken Java is not a completely new language – it has a different syntax, but it is semantically identical to Java. In fact, the language grammar that describes Spoken Java is a superset of the grammar for Java, with only fifteen extra grammar rules. Each of these additional rules maps easily onto a Java rule. This syntactic similarity makes it possible for semantic analyses based on parse tree structure to be constructed from analyses built for the original Java language without many changes.

Moving towards this more flexible input form introduces ambiguity into a domain that heretofore has been completely unambiguous. Spoken Java is considerably more lexically and syntactically ambiguous than Java. We have developed new methods for managing and disambiguating ambiguities in a software development context. In our new SPEED (SPEech EDitor) programming environment, lexical ambiguities such as homophones (words that sound alike) are generated from the user's spoken words and passed to the parser. These words, along with their lexical ambiguities and missing punctuation, form a program fragment. Our new XGLR parser (described in Chapter 5), can take this fragment and construct a collection of possible parses that contains all of its possible interpretations. Next, we exploit knowledge of the program being written to disambiguate what the user spoke and deduce the correct interpretation. Using program analysis techniques that we have adapted for speech, such as our implementation of the Inheritance Graph (described in Chapter 6), we use the program context to help choose from among many possible interpretations of a sequence of words uttered by the user. When this semantic disambiguation analysis results in multiple legitimate options, our editor defers to the user to choose the appropriate interpretation.

We conducted a user study to understand the cognitive effects of spoken programming, as well as to inform the design of the language and editor. We asked several professional Java programmers with many years of experience in software engineering to use SPEED to create and edit small a Java program. We ran two versions of the same study, one with a commercial speech recognizer, and one with a non-programmer human speech recognizer. We found the programmers were very quickly able to learn to write and edit code using SPEED. We anticipated that programmers would dictate literal code as often as they would use other forms of code entry and editing, but found that they preferred describing the code using code templates; the programmers were reluctant to speak code out loud. We expected that programmers would find speaking code to be slower than

typing; this hypothesis was confirmed. Finally, programmers all felt they could use SPEED to program in their daily work if circumstances prevented them from using their hands (RSI, hands-free situations).

1.4 Dissertation Outline

In Chapter 2, we describe in detail studies that we conducted to learn how software developers would program out loud. We explored first how program code might sound if spoken, and found that there are many ambiguities reading code out loud as well as transcribing spoken code onto paper. Next, we looked at voice-based document navigation techniques and discovered that most of them are tedious to use and have a high cognitive load. We created an auto-scrolling navigation technique that did improve cognitive load, but had usage problems of its own. The lessons we learned in this study informed the design for our new program navigation techniques. We end with an overview of the design requirements for program authoring, navigation and editing.

In Chapter 3, we present our Spoken Java language. We discuss how it differs from Java, and how it is similar. We provide some examples of Spoken Java code, and how it compares with its Java equivalent. Appendix B contains the complete lexical specification and grammar for Spoken Java. Finally, we describe a Java to Spoken Java translator that can take a Java or Spoken Java program and convert it to the other form. This is necessary to ensure that at the end of the day, a programmer's work really is Java.

Chapters 4, 5, and 6 discuss how we analyze ambiguity. These chapters form the major technology components of the dissertation. We begin with a walk-through overview of the entire analysis process. Then, we describe the XGLR parsing algorithm and framework. XGLR is an extension of Generalized LR parsing that can handle lexical ambiguities that arise from programming-by-voice, embedded languages, and legacy languages. XGLR produces a forest of parse trees that must be disambiguated in order to discover which was the one the user intended. In the last of the three chapters, we show how ambiguities arising from these lexical ambiguities are resolved in a semantic analysis engine that extends the Visibility Graph [33], a graph-based data structure designed to resolve names, bindings and scopes. Our extensions to the Visibility Graph support incremental update (for use in an interactive programming environment) and ambiguity resolution.

In Chapter 7, we introduce our SPEED Speech Editor. SPEED is an Eclipse IDE plugin that connects speech recognition to Java editing, supporting code entry in Spoken Java, and commands in our Spoken Java command language. We first introduce a sample workflow of a developer

creating and editing some code. Then we describe each of SPEED's component parts, including the Shorthand structure-sensitive editing model, and a novel user interface technique for program navigation called context-sensitive mouse grid, which was developed for this dissertation.

In Chapter 8, we present the results of two studies we conducted to learn how developers can use SPEED to program by voice. Expert Java programmers were recorded while using SPEED to build a linked list Java class. Both studies were identical, except that one study used the Nuance Dragon NaturallySpeaking voice recognition engine, and the other used a non-programmer human to transcribe the programmer's words. The study of the machine-transcribed version of SPEED showed what we can expect from the current state-of-the-art in speech recognition technology, while the study of the human-transcribed SPEED showed how good our analysis technology can be when unhampered by poor speech recognition.

In Chapter 9, we speculate on the use of speech for commenting code. By employing voice for commenting with keyboard for programming, it may be possible to overcome a likely deterrent of well-commented code: the physical interference between coding and commenting when both tasks must be performed by keyboard (or by voice recognition).

Chapter 10 concludes the dissertation. We talk about the lessons we learned while designing and building these algorithms and tools, and inform the reader of our own evaluation of the work. We end with future work, and our outlook on the future of the field.

Chapter 2

Programming by Voice

In this chapter, we address the verbalization of a programming language and explore the design space of spoken programming support in an IDE. Programming typically encompasses three main tasks, code authoring, navigation, and editing. Authoring is the creation of new code. Navigation is the browsing and reading of a program. Editing is the modification of existing code. In order to design a complete solution for programming by voice, all three of these tasks must be supported.

All three tasks share a common artifact: the program. Programs are written in programming languages, languages that are very different from the natural languages that the programmer is accustomed to speaking. This chapter describes our exploration into the following two questions:

1. Do software developers know how to speak programs out loud?
2. If they were to program out loud, what would they say?

We conducted two experiments to find out the answers to these questions. The first one asked programmers to speak pre-written code out loud as if they were directing a second-year computer science student to type in the code they were speaking. The second asked non-programmers to use commercial voice recognition document navigation tools to find slightly vague locations in a multi-page prose document. Both studies offer lessons for the design and engineering of a spoken programming environment.

In the first section of this chapter, we describe the first study and its results. We provide examples of what programmers said for different kinds of language constructs, and discuss what these results mean and what they imply about language design for a spoken programming environment. The writeup for this experiment appeared in published form [10].

The second section of this chapter discusses deficiencies in the navigation mechanisms provided by commercial speech recognition vendors. We first analyzed various navigation techniques using a GOMS model [14, 44]. We then proposed an alternate technique, which we tested with experienced voice recognition users in our second experiment. We learned several lessons about the design of voice-based interfaces, especially that speed of execution and the cognitive load are the most critical features in the design of any navigation technique.

Finally, we revisit the three kinds of programming tasks and present a range of possible designs for their spoken interfaces.

2.1 Verbalization of Code

Many possible verbalizations of written text are amenable to speech recognition analysis: simply spelling out every letter or symbol in the input, or speaking each natural language word, or describing what the text looks like, or paraphrasing the text's meaning. Spelling every word and symbol or describing the text is tedious and requires prescriptive input methods to which humans would find it difficult to conform [87]. On the other hand, excessively paraphrasing or abstracting the meaning of written content may leave too many details unspecified and even be incomprehensible to an expert.

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

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

and predictable mapping to the original language?

Our goal is to enable input that is natural to speak, but at the same time formal enough to leverage existing programming language analyses to discern its meaning. This point in the design space retains some ambiguity, but limits it so that analysis of the language is still feasible. Note that our work is *not* about programming in a natural language using natural language semantics [60, 61], but is about using features of natural language to simplify the verbal input form of a conventionally designed programming language.

2.1.1 How Programmers Speak Code

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

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

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

The recordings were transcribed with all spoken words, stop words, and fragmented and repeated words. Words with multiple spellings were written with the correct spelling according to the semantics of the original written program. Transcription took about five hours for each hour of audio tape.

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

Spoken Words Can Be Hard to Write Down

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

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

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

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

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

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

Written Code Can Be Hard To Say

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

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

One Utterance Represents Many Structures

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

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

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

Usually, only one element of a pair of matching punctuation symbols was verbalized. For example `array[i]` was expressed as "array sub i." This matches well with the mathematical rendition $array_i$. But, Java requires brackets on either side of the subscripted index. Hence, "sub" can indicate the left bracket, but there is no verbalization of the right bracket. Ending a while loop was verbalized as "close while," but no words indicated the open brace at the beginning of the while loop body. Single-line comments were demarcated at the beginning by "begin comment," but not demarcated at the end (where a carriage return would indicate the end). Multi-line comments, however, were always demarcated at both ends. In many instances, the close brace ending a block would be conflated with the beginning of the next construct; the speaker might say "and then we have a new method," or "next method."

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

Abstraction is Natural

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

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

Prosody Disambiguates

Vocal expression is as important as it is in natural language: Speakers use prosody (volume, timbre, pitch, and pauses) and vernacular to convey meaning. Prosody was used to distinguish between similar-sounding program constructs, for example, "array sub i plus plus" could mean `array[i]++` or `array[i++]`.

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

The semantic use of prosody is limited mostly to native English speakers; many non-native English speakers who speak English typically use the prosody of their native language, in which pauses, in particular, do not hold the same meaning. In our experiment, we interviewed Indian and Chinese graduate students who were non-native speakers, and none of them used the same prosody as the native English speakers. It would be interesting to see whether there are speakers of other languages who are able to employ pausing in a way that could be used for programming.

Many coding situations do not involve simple code dictation by sight, but code composition on the fly. We have designed a new study to look at how programmers speak code spontaneously when asked to write a solution to a coding exercise. This is described further in Section 8.

2.2 Document Navigation

Existing commercial tools' support for voice-based navigation provides a substandard replacement for those whose disabilities prevent them from using the keyboard and mouse. With few exceptions ([48]), all of the research exploring deficiencies in speech recognition has concentrated on dictation, which has prompted commercial speech recognition manufacturers to improve

it. However, the process of navigation in documents (or computer programs) has not been subject to similar consumer pressure, and as as a result, has stagnated. Current methods of document navigation are cumbersome and difficult to use.

Through our analysis of documentation navigation in Section 2.2.2, we show that navigation by speech is limited by two main components:

1. **Speech recognition performance** which slows down interactive use.
2. **Cognitive load** of the task as presented to the user.

We hypothesize that reducing the number of spoken commands (which reduces the total time spent in speech recognition), and reducing the number of actions that require willful thought (which reduces the cognitive load of the task) will make it possible to create a faster and easier-to-use navigation by voice mechanism.

We have designed, implemented, and tested several ways to increase the speed and utility of speech-based document navigation methods, while reducing the cognitive load at the same time. Our tool, SpeedNav, is designed to address the current situation by enabling users to navigate via an auto-scroll mechanism, reducing both the number of commands spoken (incurring less delay) and the cognitive load (enabling the user to focus more on scanning the text for the desired target than on issuing navigation commands). We present the results of our user study looking at these techniques.

2.2.1 Navigation with Commercial Speech Recognition Tools

The primary use of commercial speech recognition tools is for the creation and maintenance of text documents. A user begins by dictating into a microphone, whereupon the speech recognizer translates the speech into text for later insertion into a word processor. Speech recognizers often support some automatic formatting (such as capitalizing proper nouns and the beginning of sentences), but usually require the user to explicitly verbalize punctuation and document formatting commands (such as boldface, italics, etc).

Once the user has finished dictation, he must use the voice recognizer to navigate and edit his document. All commercial speech recognition packages support similar interfaces for these tasks. A user can say a short phrase to cause the cursor to move or to perform an editing action (cut, copy, paste, boldface, italics, etc).

Navigation commands usually involve no document content at all (e.g. move down 2 lines, go to the previous page). We call this kind of cursor movement *relative navigation*. Nuance's Dragon NaturallySpeaking supports Select-n-Say, where a user is able to augment a navigation command with an explicit phrase from the document (e.g. Select the sentence that starts with 'The quick brown.') in order to speed the process. In addition, all packages support mouse grid, a means of addressing a pixel location on the screen using a hierarchical 3x3 grid that is overlaid on the screen. Mouse grid usually requires five to six commands (one to bring it up, three or four to navigate to the target, and a final one to select it). We call these kinds of cursor movement *direct navigation*.

These designs are naive and have several flaws. In relative navigation there are too many words to speak; using the keyboard is much faster. Average typists can type around 5 keys per second [14]. In addition, the words require the user to estimate cursor distances to the desired screen location. If this location is more than some threshold lines/characters away, the user has to guess the distance, or spend time to explicitly count the distance. If the location is off the screen, the user must jump long distances and then correct for any over or undershoot. In addition, there is no auto-repeat support in commercially available speech recognizers, as there is on a keyboard. The user must repeat the navigation phrases over and over again until the desired location is found.

Select-N-Say requires the users to read and understand the text on the screen before uttering their navigation commands. This incurs more cognitive load than using the keyboard or mouse to move the cursor. Assume the user reads at a rate of 260 words per minute [14], and the average search phrase is 5 words long. If the user has already spotted the phrase on the screen, it will take him a little over one second to read the phrase. At a dictation speed of 80 words per minute, it will take another three seconds to speak it, for a total of four seconds of activity for the task. The time it takes to verbalize a location uniquely and accurately is thus far greater than the time needed to point to it with a mouse.

In addition, Select-N-Say only works when the location is on the screen and visible. If the words at the desired location are not unique, the user must include nearby but unrelated words to form a unique search phrase. If the user finds that the words he wants to speak are difficult for the speech recognizer to recognize reliably (such as the homophones: to, two, too and 2), he must either avoid speaking them in his search, or suffer the slowdown due to arduous dictation correction facilities.

This analysis of document navigation techniques is high-level. In the next section, we undertake a more formal analysis.

2.2.2 Analyzing Navigation Techniques

In this section we adapt standard formal analysis methods used by HCI researchers to study various navigation techniques used to move through a document. These techniques are activated by keyboard, mouse or voice, making them difficult to compare directly against one another. By employing formal analysis, we can use the same model to describe each technique and gain an understanding of the length of time each takes and the number of commands each takes to execute (which gives an indirect measure of cognitive load).

A look at cursor navigation techniques brings to mind Fitt's law [31], which states that the time it takes for a person to point at a location in space is

$$time \propto log(\frac{2 \times distance}{target\ size})$$

This work has been extended by MacKenzie and Buxton [62] to the action of pointing at a target on a computer screen with a mouse. In addition, later work has explored moving a mouse manually along an on-screen path to develop the Steering Law [1].

$$time \propto \frac{distance}{width\ of\ path}$$

Karimullah and Sears [48] studied cursor navigation using voice commands to move the cursor to an on-screen graphical target. Unique to their study, Karimullah and Sears enabled the users to control the cursor's velocity rather than its position.

While each of these techniques appears to approximate the document navigation task, there are important differences. Our task is multi-page; the target of navigation is usually not initially visible, and must be scrolled into view. Second, our task involves scanning for a target phrase in the midst of a page of text, not merely spotting a sole graphical target. Thus, Fitt's law and its extensions are not applicable for our task. Therefore, we see the need to develop a new model of text document navigation using speech recognition.

GOMS Analysis for Navigation

GOMS is a usability modeling technique [14] for describing human performance on a task. We use GOMS, which stands for Goals, Operators, Methods, and Selection Rules, to model how much time various common document navigation techniques will take. In particular, we use the Keystroke-Level Model (KLM) variant of GOMS [45], to illustrate how the keyboard, mouse,

$$d_{lines} = \sum_{-n \leq i \leq n} i c_{l_i} \tag{2.1}$$

$$t_{scan_n} = \alpha n + \beta \sum_{-n \leq i \leq n} c_{l_i} + \gamma \sum_{-m \leq j \leq m} c_{s_j} \tag{2.2}$$

$$t_{nav} = (1 + \rho_{error}) \sum_{-n \leq i \leq n} c_{l_i} (\Delta_{rc} + t_{scan_i}) \tag{2.3}$$

d_{lines}	=	distance in lines to the target
Δ_{rc}	=	computer's recognition delay per command
c_{l_n}	=	number of commands to scroll n lines
c_{s_j}	=	number of commands to speed up or slow down to speed m
α, β, γ	=	multipliers of components of t_{scan_n}
t_{scan_n}	=	time to scan n lines to look for the target
ρ_{error}	=	the sum of voice recognition and user error rates
t_{nav}	=	total time to navigation from start to target

Figure 2.1: Supporting equations for the GOMS model for document navigation.

and voice-activated operators involved in these techniques combine to form a complete timing measurement. The following numbers apply only to experts in both keyboard-based and voice-based navigation techniques. We are directly comparing the two in order to gain an understanding of the disadvantages afforded by the current voice-based techniques on those who cannot use keyboard and mouse.

The most important factor in the task of searching through a text document is not distance to the target (especially since the target is not often on the screen), but instead how recognizable the target phrase is. This is related to what the user is looking for, what the actual words are, how unique they are, how fast the user can read and comprehend the text, and especially whether or not the user knows the exact wording of what he is looking for or has only a vague knowledge of its contents.

In Figure 2.1, we describe the equations that govern the time it takes to navigate n lines in a text document to a desired target.

Equation 2.1 shows that the number of lines a given command scrolls (e.g. down arrow scrolls one line, page down scrolls 24 lines) multiplied by the number of times that command was given equals the number of lines traveled.

Equation 2.2 shows that the user's scanning time is proportional to the number of lines read and the number of scrolling and speed changing commands given (each command may take

the user's mind off the scanning process)[1].

Equation 2.3 shows that the total navigation time is equal to the summation for all commands that move n lines, times the delay in recognition, plus the time the person needs to scan the text after scrolling. This summation is then multiplied by one plus the expected error rate (recognizer error and user error).

Let us look at each type of navigation and run it through the equation. Given a navigation that is d_{lines} away, and a set of commands that enables you to navigate by any number of lines or any number of screens (equivalent to some number of lines), and given a target phrase-human combination that imposes a definite effect on t_{scan_n}, we can vary some variables in the following navigation methods:

- **Keyboard navigation**: Δ_{rc} (the recognition delay for a command) is very small, around 70 ms [14]. The error rate is close to zero. $c_{l_1} = 1$, $c_{l_{24}} = 1$ (assuming 24 lines per page), and for $2 \leq n \leq 23$, $c_{l_n} = n$. Auto-repeat on the keyboard lowers successive Δ_{rc}'s to 33 ms (assuming a 30 cps repeat rate).

- **Speech-based navigation by discrete jumps**: Δ_{rc} for speech recognition using IBM ViaVoice on a Thinkpad T20 P3-700/512 is 750 ms. The error rate is around 5%. c_{l_n} for any n = 1 (using the command "go down n lines").

- **Find Dialog by keyboard**: People use the find dialog only when they know the exact word(s) that they are looking for. Equation #3 is not representative of the find dialog. The following equation more closely approximates the task time.

$$t_{find} = \quad c_{find}\Delta_{rc} + t_{typing} + c_{OK}\Delta_{rc} + (E(P_{find}) - 1)(c_{next}\Delta_{rc} + t_{scan}) \qquad (2.4)$$

Users operate the find command by first issuing a command to open the find dialog, then typing in the words they are looking for, and hitting the OK button. For each successive search result highlighted by the system, the user must scan the line to see if the desired words were found; if not, the user issues a find-next command and repeats. The expected number of times to repeat is half of the number of times the words appear in the target document, assuming that the user's target is uniformly distributed amongst the search results.

[1] See discussion in Section 2.2.3

In this mode, Δ_{rc} = 70 ms, c_{find} = 1, c_{OK} = 1 and c_{next} = 1.

- **Find Dialog by Voice**: This is the same as Find Dialog by Keyboard, except that Δ_{rc} = 750 ms, and t_{typing} is replaced by $t_{dictation}$, the time it takes to dictate (with errors and error correction) the target phrase into the find dialog. c_{find} is usually 2 (Edit menu – Find menu item), but can be 1 with a speech macro. c_{OK} = 1, and c_{next} is usually 2 (Edit menu – Find Next menu item), but can be 1 with another speech macro.

- **Select and Say**: This is similar to the find dialog, but the target phrase must be visible on-screen, thus we must add in the scrolling time to make the line visible on-screen to the equation. The equation is as follows:

$$t_{selectandsay} = t_{scroll} + t_{scan} + t_{dictate} \tag{2.5}$$

  t_{scroll} is the time to scroll within one screen of the target using any of methods described above. t_{scan} is the time it takes for the person to find the desired target on the screen (related to a user's skimming capability) and is inversely related to the target's uniqueness. Once the person finds the target, they must speak it out loud ($t_{dictate}$) and then the software highlights the phrase.

As one can see, the two dominant controllable factors here are t_{scan} and c_{l_n}, the scanning time and the number of commands issued by the user. Reducing either of these numbers should result in faster navigation times.

2.2.3 Design

The design phase for SpeedNav started with a survey of expert voice recognition users for their impression of existing voice recognition packages, with emphasis on their use of the editing facilities.

Expert Interviews

We interviewed three experienced users of speech recognition – those who use speech recognition for most of their work during the day. They were professionals (non-programmers) who use computers in their daily work, but due to motor disabilities, employ non-keyboard and non-

mouse-based means of using the computer. Each had used voice recognition extensively. One was still using voice recognition as his primary mode of manipulating the computer.

We asked the experts about their experiences using voice recognition software, including what they used it for, the kinds of training they needed to make the input method usable, and the problems they encountered while doing so. Several issues stood out amongst all the interviews:

1. The speed and quality of voice recognition was always described as too slow, too cumbersome to use, too unreliable, bad at recognizing accents, and error-prone. These were the biggest issues by far. When described in terms of frustration, one particularly good user said he experienced 2-3 frustrating moments per hour. Usually if a user had the use of his hands, he would "cheat" and revert to using a keyboard whenever voice recognition began to fail him.

2. Experts use voice recognition in different ways. Some use it only for dictation, and perform editing by hand. If any editing by voice was performed, it would be a short period of time after dictation (such as completing a paragraph). Many find that it does not work in all needed applications, and does not work in technical applications (e.g. for computer programming).

3. People had problems editing by voice. In particular, they could not easily verbalize where they wanted the cursor to go, nor could they easily figure out how to command the cursor to go there. For some, using the mouse grid feature was the only reliable way they could move the cursor to a location they could see on the screen. All those we interviewed recognized that speech macros could be useful to speed up this task, but most did not use them.

Prototype Ideas

We developed many ideas towards achieving our goals. We prototyped six of them in Microsoft Word using Visual Basic. These prototypes were useful in furthering our understanding of the difficulties in designing a good navigation tool, and we present our findings here.

Our major idea was auto-scrolling with a voice-enabled speed control. To reduce the number of commands the user must say, the user starts to scroll the document in a particular direction, scanning the document as it scrolls by for the desired phrase. He stops the scroll when he sees his target on the screen. We assert that this mechanism also reduces the cognitive load to scan the text as it scrolls by – the user does not have to re-utter the text in order to select it. We also add a speed control, enabling the user to match the scrolling rate to his natural scanning capability.

To support our idea, we extend the GOMS analysis (from Section 2.2.2) for the following two methods: auto-scroll and auto-scroll with speed control.

Speech-based navigation by auto-scroll: In this mechanism, there are only two commands necessary, "start" and "stop", so $c_{l_{d_{lines}}}$ = 2, and for all other n, c_{l_n} = 0. The time between successive scroll actions is t_{pause}. This is set to a particular value (the next mechanism enables speed control) but must be greater than the t_{scan} or the person will not have enough time to comprehend the text before it scrolls away. Δ_{rc} = 750 ms, so the drop in the number of commands to a constant two commands should have a significant effect to lower the overall navigation time. The velocity of the scrolling motion directly affects the precision of the user to avoid overshooting and indirectly affects accuracy of cursor placement.

Speed-based navigation by speed-controllable auto-scroll: Finally, we take the above mechanism and add two commands to control the pause time: "faster" and "slower". We hypothesize that the desired navigation speed of the user will follow the equations below:

$$t_{start} = \Delta_{rc} \tag{2.6}$$

$$t_{stop} = t_{scan_n} + \Delta_{rc} \tag{2.7}$$

$$t_{pause} \geq t_{stop} \tag{2.8}$$

t_{start} = time it takes for the system to react to a start command

t_{stop} = time it takes for the system to react to a stop command

t_{pause} = time between successive actions by the editor

Once the user begins scrolling, the system sets the pause time to the user's personal initial value. The user may issue the speed control commands to increase and decrease this pause time, but may not lower the time below the time he needs to scan n lines of text for the target. The pace of scrolling will drop (pause time will increase) as the user nears the target. While the Δ_{rc} is 750 ms for the "start" and "stop" commands, we feel it is possible that the "faster" and "slower" commands overlap the pause time, therefore its contribution to t_{scan} is zero (This finding requires further study). Most users do not realize that Δ_{rc} is so long, and therefore they do not slow down in time, and overshoot the target. This overshoot problem will diminish (but never go away) with practice.

Prototype Results

We implemented several variants of the above idea. The first was to scroll the screen using the cursor, but that caused the eye to follow the cursor instead of the text. We then tried a smooth scroll without the cursor, but found that smoothness caused blurring of the text, making it hard to read at all scroll speeds. The third was a line-by-line scroll, but still found that the text was too jerky to read. Thus, we rediscovered a principle of perception, the human eye cannot read text that is constantly moving. The eye must fixate on a word in order to read it [14].

The fourth prototype switched to auto-scrolling by page instead of by line; the user then controlled the length of the pause between the scrolling operations. It was important to scroll less than a page at a time so that the user could recognize continuity in the scroll by seeing words that had been at the bottom of the page, now at the top. We also learned that after the scroll is complete, it is important to leave the cursor in the middle of the screen, rather than at the bottom, to minimize the amount of further cursor motion needed to reach the desired character position.

Final Design

The final design for SpeedNav incorporates all of the lessons learned above. The main feature is an auto-scroll and pause, scrolling the text of the document rather than the cursor. The scanning speed is controlled by the pause time, initially set at 2 seconds. Each page-down/page-up action is invoked over a period of 100 ms, and scrolls 3/4 of the page, creating a 1/4 page overlap between screens of text. We added line scanning (line up and line down) which operated solely by scrolling and had a variably controlled speed between 2 lines per second and 20 lines per second. In addition, we added character scanning (left and right) with variably controlled speed initially set at 8 characters per second. When the user switched from page scrolling to line or character scrolling, the cursor position was placed in the middle of the screen horizontally and 1/3 down the page vertically.

We supported nine commands in two categories:

1. **Navigation:** Page Down, Page Up, Go Down, Go Up, Go Left, Go Right
2. **Speed:** Faster, Slower, Stop

Most of the commands are two words/syllables to aid in speech recognition accuracy.

We experimented with a compensation for cursor overshoot. When the user stopped the cursor, we knew its velocity, and could estimate the amount of Δ_{rc}. We used this information

to calculate how many units (characters, lines, or screens) we would automatically backtrack the cursor. Unfortunately, in our preliminary experiments, users would "game" the system, trying to estimate the amount of cursor overshoot and anticipating when to stop it, which directly interfered with our naive implementation of this option. It was removed from SpeedNav in the final study.

2.2.4 Implementation

We developed two implementations of SpeedNav. Our hardware platform for both was an IBM Thinkpad T20 with a Pentium III running at 700 MHz and 512 MB of RAM running Windows XP Pro. Both implementations used IBM ViaVoice 9 as the speech recognition software. IBM ViaVoice provides an API to the programmer called SMAPI (Speech Manager API) which enables an application to access voice recognition services, including access to dictation and command and control grammars.

Our first implementation was written in Visual Basic, which we used to script the Microsoft Word 2000 word processor. The speech recognition interface was implemented through IBM's ActiveX controls (provided in ViaVoice 8). The second implementation was written in Java using a modified Swing Stylepad word processor. We interfaced to the speech recognizer via an IBM-provided JSAPI (Java Speech API) plugin.

Technology Problems

We encountered several problems with the Visual Basic implementation of SpeedNav. First, we found the Visual Basic language (which we learned for this project) to be quite a bit harder to understand and use than a more traditional programming language like Java. In addition, the COM OLE Automation documentation for Microsoft Word is poorly organized (alphabetically by function name, even though it is an object-oriented API) and at crucial times, the online web-based documentation was unavailable. We found that our control of MS Word was superficial and we could not implement sophisticated shading behaviors that we had intended for a seventh prototype. In addition, all of Microsoft Word's scrolling techniques left the cursor at the bottom of the screen as it scrolled. We did not have enough control over this to move the cursor to a stable screen position.

By contrast, Java's open source implementation and documentation made it possible to work around any difficulties we had in massaging its software to behave the way we intended. However, Java's less than speedy performance prevented us from implementing smooth scrolling behaviors (some of MS Word's scrolling functions were also quite jerky, but it was possible for the

		# Lines	Total Time (sec)	Multi-page Time (sec)	Reading Time (sec)	# Cmds	# Recog Errors	# User Errors	# Mouse Grid
User 1									
Doc 1	Task 1	87	235	226	74	42	5	3	0
	Task 2	55	75	68	42	5	1	0	0
	Task 3	59	162	150	60	17	3	0	0
	Task 4	92	86	84	48	11	1	0	0
Doc 2	Task 1	88	65	48	22	9	2	1	0
	Task 2	62	129	118	81	9	0	0	0
	Task 3	133	235	208	190	12	1	2	0
	Task 4	154	290	271	173	22	2	4	0
User 2									
Doc 1	Task 1	87	61	58	41	13	1	0	1
	Task 2	55	18	11	10	5	1	0	1
	Task 3	VOID	VOID	VOID	VOID	VOID	VOID	VOID	VOID
	Task 4	92	131	129	85	19	0	1	2
Doc 2	Task 1	88	59	26	N/A	8	0	0	0
	Task 2	62	61	47	N/A	11	0	0	0
	Task 3	133	205	188	N/A	27	0	1	0
	Task 4	154	250	250	N/A	27	0	1	0
User 3									
Doc 1	Task 1	87	69	61	26	16	0	3	0
	Task 2	55	63	51	47	8	0	1	0
	Task 3	59	83	74	58	11	0	1	0
	Task 4	92	100	99	57	23	0	1	0
Doc 2	Task 1	88	33	24	N/A	5	0	1	0
	Task 2	62	153	127	N/A	27	0	4	0
	Task 3	133	98	98	N/A	12	0	4	0
	Task 4	154	145	135	N/A	17	0	1	0

Table 2.1: This table shows the data collected from the users in our study on document navigation. (User 2, Task 3 is voided because he got confused about which target he was looking for. Reading times for Tasks 5-8 (using SpeedNav) for Users 2 and 3 are marked N/A because we could not reliably differentiate between the time spent reading and the time spent issuing commands.)

most part to avoid them), but due to time constraints, we were forced to accept it.

We eventually abandoned the Microsoft implementation in favor of the Java version, and used it in our user study, described in the next section.

2.2.5 User Study

In this section, we describe a study of three expert users of voice recognition. We asked each user to perform several editing-by-voice tasks using their own voice recognition tools and our implementation described above.

Hypothesis

We predict that the speed of the tasks will improve from the user's own voice recognition tools to our auto-scrolling cursor. In addition, the number of commands spoken should drop, reducing the delay caused by the speech recognizer response time. However, due to the need to control the speed of the cursor, there may be more commands than we predict using our GOMS analysis. We also anticipate the cognitive load of the navigation tasks will go down, as measured the amount of time it takes for the user to scroll the document to the page containing the target.

Methods

We asked expert users of voice recognition software to perform 8 tasks divided into two similar groups of 4. Each group involved short, medium, long distance, and backwards medium distance navigation through a 10-15 page scene from a Shakespeare play (Romeo and Juliet, Act III, scene 1, and Taming of the Shrew, Act II, Scene 1). One user was familiar with the plays, while the others were not.

Each task was phrased as a search for a specific line in the play. However, learning from our pilot study, we did not give the user the specific words but only a vague description of the line. This approach seems to better approximate the kind of navigation task a user is likely to perform on documents that are unknown to him, or those that are not fresh in his mind. This also simulates the kind of search we envision that programmers would employ, since they often know what they are looking for without knowing the exact wording of the code.

The first group of tasks was performed using the expert users' own voice recognition setup, with their own equipment and software (User #1 used Dragon NaturallySpeaking 5.0 on a P2/450 128MB. User #2 used Dragon NaturallySpeaking 5.0 on a P3/550 320MB. User #3 used Dragon Dictate 3.01 on a P3/500 128MB). The second group of tasks was performed using our SpeedNav software and a laptop that was brought to each session. Users were trained for 10 minutes on IBM ViaVoice using the ViaVoice User Setup Guru. Users then trained for about 5 minutes with SpeedNav on another sample document to gain a feel for our software. We provided a cheat sheet with a list of the nine SpeedNav commands to each user during their tasks.

We video-recorded each session for later analysis.

At the end of the study, we interviewed the participants to gauge their opinions and feelings comparing the two navigation methods.

Metrics

		$\frac{\#\ Commands}{\#\ Lines}$	$\frac{Multi\text{-}page\ Nav\ Time}{\#\ Lines}$
User 1			
Doc 1	Task 1	0.48	2.60
	Task 2	0.09	1.24
	Task 3	0.29	2.54
	Task 4	0.12	0.91
Average		0.25	1.82
Doc 2	Task 1	0.10	0.55
	Task 2	0.15	1.90
	Task 3	0.09	1.56
	Task 4	0.14	1.76
Average		0.12	1.44
User 2			
Doc 1	Task 1	0.15	1.67
	Task 2	0.09	0.20
	Task 3	VOID	VOID
	Task 4	0.21	1.40
Average		0.15	1.09
Doc 2	Task 1	0.09	0.30
	Task 2	0.18	0.76
	Task 3	0.20	1.41
	Task 4	0.18	1.62
Average		0.16	1.02
User 3			
Doc 1	Task 1	0.18	0.70
	Task 2	0.15	0.93
	Task 3	0.19	1.25
	Task 4	0.25	1.08
Average		0.19	0.99
Doc 2	Task 1	0.06	0.27
	Task 2	0.44	2.05
	Task 3	0.09	0.74
	Task 4	0.11	0.88
Average		0.18	0.99

Table 2.2: This table shows two aggregate measures derived from our data on document navigation: Number of commands divided by number of lines read, and the multi-page navigation time (in seconds) divided by the number of lines read.
(User 2, Task 3 is voided because he got confused about which target he was looking for.)

We measured several quantities (shown in Table 2.1) to understand the impact of using traditional speech-recognition-based navigation vs. auto-scroll to navigate through a document. We measured the total time per task, time to scroll the document to the page that contained the target (called *multi-page navigation*), the total number of commands spoken, the number of recognition errors, the number of errors caused by misuse of the tool, and number of times the user invoked the

mouse grid. In addition, for the users' own speech recognition-based tool, we measured the time spent reading and scanning text (assuming that the rest of the time was spent issuing commands and waiting for the response of the speech recognizer).

Discussion of Results

We compared the two tools using two aggregate measures derived from our data (see Table 2.2). The first is the number of commands divided by the number of lines traveled. The second is the multi-page navigation time (in seconds) divided by the number of lines traveled.[2]

We ran an unpaired t test that showed no significant difference between the users' own tools and SpeedNav ($P = 0.79$ for the multi-page numbers, and $P = 0.31$ for the commands per line numbers), however, the means for SpeedNav were better. We feel that if we re-did our experiment with enough users, we would be able to better discern a difference between the tools.

Another difference between the tools is that users had years of experience with their own tools, but only 10 minutes of practice with SpeedNav. Perhaps, users more practiced with SpeedNav would perform better. From our experience, users dramatically improve their performance with voice recognition software over time.

Post-Study Interview

We conducted an interview with each study participant after completing the tasks. In general, participants said that SpeedNav was easier to use than their existing speech recognition system. There were fewer commands required to move to the desired location, and the commands themselves were easy to remember. Participants also appreciated the speed control. One participant also liked the cursor moving by character because he could control the cursor speed to match his reading speed and use the cursor as his pointer.

On the other hand, participants universally had trouble with cursor overshoot, especially when navigating to a location within the current screen. While in general, the cursor overshoot problem is an inherent component of motion-based navigation, SpeedNav exacerbated the problem by not supporting precise positioning within a page. One user wanted to be able to place the cursor at natural landmarks in the document (top of page, top of document, etc). Users observed that relative positioning of the cursor (by motion along the vertical or horizontal axis) is not always the most

[2] We use the multi-page navigation time rather than the total time because users universally found that within-screen navigation was much more cumbersome and inaccurate using SpeedNav.

efficient path to a point on the screen. One user preferred to use mouse grid exclusively to navigate within the screen, and was very proficient at it. Another user felt that if he was more familiar with the document, he would have been able to more effectively use the speed controls, and slow down before reaching the target (minimizing cursor overshoot).

2.2.6 Related Work

Our solutions are inspired by Manaris, McCauley and MacGyvers' SUITEKeys voice-activated keyboard and mouse [63], and Igarashi and Hughes's non-verbal voice input [41].

Manaris specifically addresses individuals with permanent motor disabilities (such as those who use a keyboard via mouthstick) and enables them to "press" keys on a keyboard and "move" the mouse by speaking low-level actions. It is not clear whether the voice keyboard has auto-repeat, but the voice can start the mouse cursor moving and then cause it to stop with another utterance. Inspired by this work, we created an ability to start the text cursor in motion and stop it at a later time.

Igarashi enables people to use pitch and volume (instead of speech) to control a button or joystick. We applied this idea to our solution by enabling a user to control the speed of the cursor movement (though we use words to command it rather than non-verbal communications).

Igarashi's earlier work on speed-dependent automatic zooming [40] is also relevant to our work in that he mentions that when a user scrolls too fast, it is hard to read the text, even to just get a sense of where he is. He proposes an automatic zoom-out feature as scrolling gets faster to enable people to gain a better sense of where they are. We believe that our pause concept is superior since stationary text is much easier to read than text moving at any velocity.

Karimullah and Sears [48] studied speed-based cursor control. They recruited non-expert users of speech recognition (none had any visual, heading, speech or cognitive impairments), where we are targeting our work at expert users who are using speech recognition as a primary form of input. Their users, however, did experience the ubiquitous cursor overshoot problem, even though they restricted themselves to a single cursor speed. In addition, our task is more realistic in a work setting, making the user search through a text document rather than a simple graphical target. Finally, we are working with multi-page documents, which implies that a simple Fitt's law of motion is inapplicable to navigate to the desired target.

2.2.7 Future Work

In our post-study interview, one user expressed a desire for SpeedNav to adjust its speed automatically according to the density of the document text visible on the screen (measured in visible characters). This would be even more interesting if SpeedNav controlled the initial speed of motion when beginning a navigation task as well. This would provide an alternate means of adapting to the user's inherent reading speed than provided by Igarashi's zoomable user interface. Both controls attempt to preserve a constant density of text per unit time.

Another method to speed up the user's navigation performance is to improve the user's own technique in scanning text. A course in speed reading might nicely complement our SpeedNav work.

In addition, we think that if we add a shaded region of lines to the document it could make these lines easier to read because human eyes are drawn to regions of colors that are different than their surroundings. The idea is to shade three lines of text with a pastel background, and when the auto-scroll is active, we move this shaded region down the page, one line at a time, until it reaches 3/4 of the way down the screen (it will make this journey within the pause time for reading this page). When the shaded region hits this point, the entire screen will scroll 1/2 a page (leaving the shaded region 1/4 from the top of the screen). This shaded region will help draw the eye (from the cursor) and focus the reader to scan the text from top to bottom as the shaded region moves.

We feel we can improve the sophistication of the cursor overshoot correction algorithm in a novel way. Since $\Delta_{rc} > 0$ requires that the cursor will go further than where the user intended, we propose to place two shaded regions (of different colors) on the screen. The first shaded region goes at the top of the page (when the user is scrolling down, and at the bottom when scrolling up. (For the rest of this example, we will assume a downward scroll.). The second, $\Delta_{rc} \times speed_{scroll}$ lines below it, is centered in the middle of the screen. When the user scrolls, he will read the text in the center shaded portion, but in reality the system assumes the "cursor" is in the upper shaded portion. When the user says "stop", the "cursor" in the upper shaded portion scrolls down to the center shaded region and stops, eliminating the users' perception of the overshoot. This kind of trickery was studied by Karimullah and Sears [48] for cursor motion within a screen towards a graphical target. Users experienced higher error rates with such an automatically correcting cursor, but the experiment used no speed control, which we feel might enable users to slow down to a comfortable reading speed.

2.2.8 Summary

Document navigation, the less glamorous aspect of speech recognition, deserves more attention from the research (and commercial) community. An improvement in this functionality will enable those with motor impairments to enjoy the same ease of editing that non-impaired people take for granted. In addition, program navigation is an important part of software development. Improvements to document navigation would be beneficial for programmers.

This work contributes to our understanding of the performance of the current state-of-the-art when used by people with motor impairments. We have shown through a GOMS analysis that the only ways to improve this performance are to reduce the number of commands and the cognitive load on the user. Reducing the latency in speech recognition will help, but the problems will not go away until speech recognition response is as fast as a keyboard or mouse. Our SpeedNav tool showed the potential to reduce the number of commands and the cognitive load through an auto-scrolling mechanism. Even though our results were inconclusive (comparing SpeedNav to commercially-available solutions), further development along these lines (as well as a larger user study) should show a more significant result.

2.3 Programming Tasks by Voice

In this section, we summarize what we learned from the two experiments described above and relate them directly to voice-based software development.

2.3.1 Code Authoring

Our spoken programs study showed that code authoring support requires a speech analysis system that can understand the natural language words of a program spoken out loud. Punctuation is generally omitted from speech, causing many ambiguous interpretations of the input, even for a human listener. Some of the ambiguities are lexical (words that are spelled, capitalized and separated from one another in various ways). Others are syntactic – missing punctuation creates many possible structures for short utterances. Despite the ambiguities here, however, it should be possible for a human listener who is cognizant of the entire program being written to figure out the right interpretation. What remains to be proven is that a computer can be programmed to disambiguate as well or better than a human could.

In the study, some programmers described the code they wanted to create rather than

speaking the program's literal words. For example, a programmer might say "There is a class here." Recognizing command forms of these phrases, in a form of phrase-based code template expansion, would best support this style of voice-based programming. The acceptance of this style would also confirm Snell's assertion that keyword-based code template expansion contributes to efficient input of code by voice.

Note that our spoken programs study looked at programmers speaking pre-written code that they read off a piece of paper. There was no visual or auditory feedback of their progress through the program, nor any way to verify the correctness of the program they spoke. In addition, the program was spoken linearly from top to bottom, which is different from the way most programmers create new code. Some software developers plan the interface to their code before they write the implementation; some write one function and test it before writing the next. Each of these styles would require a speech system to accept partial code or code out of context; supporting the analysis of spoken incomplete or incorrect programs is vital to a usable solution.

While we feel that we have identified the spoken language used by the study participants for code authoring, our understanding of what kinds of errors they make will require further study.

2.3.2 Code Editing

Editing code requires a mix of code authoring and commands for manipulating the programming environment. We have identified four kinds of essential commands necessary for editing a program.

1. Edit and Replace: A developer will select a piece of code and then edit the structure. Replace is similar to edit, but erases the structure and then inserts code.

2. Insert Before and Insert After: Many program structures are found in sequences (e.g. sequences of statements, sequences of class members). When editing a large structure with many component members, a programmer will select one structure in a sequence (or "epsilon", the empty sequence) and add another element in the sequence before or after the selection.

3. Delete: This one is self-explanatory.

4. Copy, Cut, Paste, Undo, Redo: These are the standard clipboard and history commands with which programmers and other document writers have long grown comfortable.

5. Text-Based Editing Commands: When a programmer needs to edit an identifier name, commands that move the cursor around (Go Left, Go Right, Go Left Word, Go Right Word, Go Home, Go End), select text (Select All, Select Left Word, Select Right Word), delete text (Delete Left, Delete Right, Delete Left Word, Delete Right Word), insert new text (any letter, Space, Insert Line), and change capitalization (Cap That, Lowercase That) are essential.

The realization of these editing commands in our programming environment will be described in Chapter 7.

GOMS Analysis for Program Editing

We can use GOMS to analyze the editing commands we have chosen for our new voice-based programming environment and see how they compare with typing. We will employ the same KLM model as we used for the document navigation analysis.

In Figure 2.2, we show the equations that govern basic entry of code by keyboard and by voice. The equations show that voice-based code entry is almost always slower than typing. The main numbers we care about are the time to enter a word on the page t_{word} and the time to enter a whole program statement t_{stmt}. For typed programs, the time to enter a statement is the number of words in the statement times the number of keystrokes to type each word. As we stated above, Δ_{rc}, the recognition delay, is around 70 ms to press a key. For spoken programs, the time to enter a statement is the number of words times the speech recognition delay modified by the recognition error rate. Using good speech recognizers, the recognition delay is around 750 ms. Even assuming that there are no speech recognition errors, each word would have to have 10 characters in it to slow typists down as much as speech recognition users. Adding a standard error rate for a trained recognizer of around 0.1%, and probably a much higher error rate for cascaded errors, the words in a program would have to be considerably long for speech to be competitive with typing.

Most of the editing commands activated by speech have equivalent single keystroke forms activated by the keyboard. Thus, we only have to multiply by the Δ_{rc} for each modality to understand the speed of the interface for code entry.

When editing pre-existing code, keyboard users merely point their cursor at the start of the text they want to edit, and start typing. Voice-based programmers must select the insertion point carefully using one of the document navigation techniques described above. Mouse grid is the fastest for pointing to a position visible on the screen, only requiring around four or five spoken commands. Since mouse grid is geometrically identical every time it is invoked, voice recognition

$$t_{word} = l_{word}\Delta_{rc} \qquad Typing \qquad (2.9)$$

$$t_{word} = (1 + \rho_{error})\Delta_{rc} \qquad Speech \qquad (2.10)$$

$$t_{stmt} = t_{word}\, l_{stmt} \qquad Typing\ or\ Speech \qquad (2.11)$$

t_{word}	=	time to enter a word on the page
l_{word}	=	length of word in keystrokes
t_{stmt}	=	time to enter a statement on the page
l_{stmt}	=	length of statement in words
ρ_{error}	=	the sum of voice recognition and user error rates
Δ_{rc}	=	computer's recognition delay per command or keystroke

Figure 2.2: Supporting equations for the GOMS model for program entry.

users who are trained do not have to wait for the recognizer to respond to their spoken numbers before uttering the next. Thus the total delay for pointing with mouse grid is very close to the delay for recognizing a single command, around 750 ms. Once the selection point is determined, voice users speak one command (Edit This) and then start dictating their changes to the code.

Note, that keyboarding editing commands like Delete Left, Delete Right, Go Left, Go Right, and the like, each only take one keystroke to activate (70 ms). Repeated keystrokes are even faster (33 ms). Voice commands for manipulating the cursor are each as slow as dictating a word, except that usually the speaker must wait for each command to activate before moving onto the next. So each command takes at least 750 ms, even, and especially, repeated commands. Thus it is likely that a voice recognition user edits code much more slowly than a keyboard user.

One technique for avoiding such delay in editing is to respeak the entire statement when editing it, or when a mistake in recognition is made. In this case, there is one command to restart (Select All); further speech will overwrite the existing statement and insert the new text. If a misrecognition can be avoided, and potentially difficult to control capitalization or word spacing is not required, then this technique can speed up code editing and dictation significantly.

2.3.3 Code Navigation

Much of a programmer's time is spent browsing and navigating through his code. Navigating code requires having a mental model of what the program does and how it is structured. Consequently, much work has gone into program visualization tools to illustrate high-level program structure and facilitate browsing [23, 109, 68, 82, 77, 64, 57, 24, 7]. When it comes to actuating

the navigation, however, IDEs rely on keyboard and mouse, especially on the ability to click on or scroll to an item of interest in order to manipulate it. Voice recognition presents problems with both of these forms of absolute and relative navigation techniques.

To study the issues in a simpler form, we looked at voice-based navigation in text (non-program) documents in the second experiment described above. Though the nature of the content is different, the need to navigate quickly to parts of the program where the semantics, but not the actual written code, is known is the same. The techniques used for verbalizing the navigation to these kinds of locations in a document are also very similar. Autoscrolling is one technique that can be used to quickly browsing through code, but the overshoot problem will need to be fixed before that becomes a good solution.

Inspired by our navigation exploration, we came up with two new navigation techniques which we think are much better than previous approaches and quite suitable for programmers as well. These are context-sensitive mouse grid and phonetic search. We have developed a mouse grid that is sensitive to program structure, enabling programmers to hierarchically navigate to the desired program element quickly and naturally. This is described in Section 7.3.4. Phonetic search is also useful, when extended with the ability to discover phonetically similar abbreviated words, such as identifiers often used in programming. This technique is described further in Section 7.3.8.

Chapter 3

Spoken Java

3.1 Spoken Java

In the previous chapter, we described two studies we conducted that explored how people using voice recognition might approach the tasks of code authoring and navigation. The goal of these studies was to inform the design of a new naturally verbalizable alternative to Java that we call Spoken Java. Spoken Java is a dialect of Java that has been modified to more closely match what developers say when they speak code out loud. Spoken Java is the input form for our new programming environment called SPEED (SPEech EDitor). SPEED is an editor for Java programs that allows voice recognition users to compose and edit programs using Spoken Java. Spoken Java code is ultimately translated into Java as it appears in the program editor. Spoken Java is designed to be semantically equivalent to Java – despite the different input form, the result should be indistinguishable from a conventionally coded Java program.

Java was chosen as the prototype language for a number of reasons. First, it is in widespread use in both industry and academia. Many people are learning Java and programming real applications in it. While Java is a large language, it admits tractable static analyses to discover the meaning of entire programs. Other popular languages are known to be more difficult to analyze (e.g. C and C++). Finally, Java is representative of programming languages in general. The knowledge we gain and methodologies developed by prototyping our speech system with Java can easily be applied to other statically analyzable programming languages.

Several features of Spoken Java were added to address the concerns brought up during our study of code verbalization. Most punctuation is optional, and all punctuation has verbalizable equivalents. Each punctuation mark may have several different verbalizations, both context-

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

(a)

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

(b)

Figure 3.1: Part (a) shows Java code for a for loop. In (b) we show the same for loop using Spoken Java.

insensitive (e.g. "open brace") and context-sensitive (e.g. "end for loop"). We have reversed the phrase structure for the cast operator to better fit with English (e.g. "cast foo to integer") and provided alternate more natural language-like verbalizations for assignment (e.g. "set foo to 6") and incrementing or decrementing a value (e.g. "increment the i^{th} element of a" in place of "a sub i plus plus").

Figure 3.1 shows an example of how a Java program might be entered in Spoken Java (carriage returns in Spoken Java are written only for clarity). Note the lack of punctuation, the verbalization of operators (`less than` and `equals`), an alternate phrasing for assignment, and the verbalization of the `cos` abbreviation. (The example assumes the correct spelling for `x` and `i`).

Figure 3.2 illustrates more program structure. Note the lack of capitalization, separation of words `to` and `buy,` (and `print` and `line`), the assumed correct spelling for every word (which should not be assumed as the user speaks the code), the expansion of the abbreviation `ln` to `line`, the optional punctuation character `dot`, and the overall lack of braces and parentheses. Also take notice of the lack of a right parenthesis or suitable synonym after `thing to buy` in the method declaration parameter list.

3.2 Spoken Java Specification

Spoken Java is defined by a lexical and syntactic specification in the XGLR parsing framework described in Chapter 5. Motivated by the language used by the programmers in the study, the lexical specification supports multiple verbalizations by allowing many regular expressions to map to the same token. The grammar is similar to a GLR [95] grammar for Java, but contains fifteen additional productions to support four main features: a) lack of braces around the class, interface, and other scoped bodies, b) different verbalizations for empty argument lists as opposed to lists of at

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

(a)

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

(b)

Figure 3.2: Part (a) shows Java code for a Shopper class with a shop method. In (b) we show the same Shopper class and method using Spoken Java.

least one argument, c) an alternate phrasing for assignment and d) an alternative phrasing for array references. Each of these additional productions naturally maps to a structure in the Java grammar.

The Spoken Java language is presented in its entirety in Appendix B.

Spoken Java is considerably more ambiguous than Java, mainly due to lexical ambiguity and the lack of required punctuation in the language. Some lexical ambiguities arise due to English words being used with multiple meanings. For example, “not” could mean both boolean inversion and bitwise inversion. “Star” could mean both multiplication and Kleene star. “Equals” can stand for assignment and equality. “And” can be boolean and, bitwise and, or a substitute for comma in a sequence of parameters or arguments. Many words can be both Spoken Java keywords and identifiers; a few that are specified are “array,” “set,” “element,” “to,” “new,” “empty,” “increment,” and “decrement.” The full set of alternatives supported by Spoken Java can be found in the lexical specification in Appendix B.1. These ambiguities are identified after the voice recognizer returns them to the analysis engine.

The lexical specification also contains several ways to say each construct, to cover the range of expression we found in the Spoken Programs study. For example, for the Java `instanceof`, one can say “instance of” or “is an instance of.” Java’s >> operator can be spoken “right shift” or “r s h.” Java’s == can be spoken as “equals,” “equal equal,” or “equals equals.”

These alternate utterances often provide unambiguous ways to say a particular token when the original word could be construed to have multiple meanings.

A few of the keywords and numbers in the Java lexical specification are homophones with other words, usually identifiers. For instance, "to" is a homophone with "2," "too" and "two." "One" is a homophone with "1," and "won." "char" can also be spelled "car." "4" can be spelled "four," "for" or "fore." We run each spoken word through a dictionary of homophones to generate all possible spellings for a word before analyzing its meaning.

As reported by an XGLR parser generator, Spoken Java contains 13,772 shift-reduce, reduce-reduce, and goto-reduce conflicts. In addition, each of the 59 lexical ambiguities causes a form of shift-shift conflict in the parser, which brings the total number of conflicts to 13,831. By contrast, Java contains only 431.[1] Each conflict results in a runtime ambiguity and a slight loss of performance away from linear time. In Java, each of these runtime ambiguities is resolved within one or two tokens (due to the language design); by the time the entire program has been successfully parsed, there are no ambiguities in the parse. In Spoken Java, however, many of those ambiguities survive parsing, requiring further analysis to identify their meaning.

All of the analyses used to generate, propagate and resolve ambiguities will be presented in Chapters 5 and 6.

3.3 Spoken Java to Java Translation

Spoken Java was designed to be only the input form of a spoken programming environment; in order to be accepted by colleagues and co-workers, a programmer's end product must be traditional Java code. Thus, it was designed to have a similar syntax to Java in order to make it easy to translate back and forth. We have developed a grammar-based translator that can take an unambiguous parse tree for Java or Spoken Java and convert it to a string in the other language. As will be described in Chapter 7, the SPEED program editor employs the translator to convert the programmer's speech into Java, and to convert the Java program itself into Spoken Java for editing or training purposes.

This translator is based on a grammar-oriented specification. For each production in the Java and Spoken Java grammars, a list of operators is specified to define how to translate

[1] By comparison, C++, a language with considerably more syntactic ambiguities than Java, has 2,411 conflicts. It is impossible to predict the increase in the number of conflicts in the spoken version of C++, since each language's spoken variant must be designed specifically according how it is spoken in the vernacular, and not with a standard calculation.

each right-hand side (RHS) symbol. There are 19 operators, most (16) dealing with the terminals in the grammar and only three dealing with nonterminals. Nonterminals are simply decomposed into their component RHS symbols (HANDLE_KIDS). Sequences can be stripped of separators or have them added (COPY_LIST_DROP_SEPARATOR, COPY_LIST_ADD_SEPARATOR). Terminals can be copied (COPY_LEXEME), added (ADD_LEXEME), dropped (DROP_LEXEME), replaced (REPLACE_LEXEME), or converted to their default representation in the other language (DEFAULT_LEXEME). Multi-word identifiers in Spoken Java are translated by concatenating the words together in one of three styles: CamelCase (words with initial capital letters joined by concatenation), C++ (words joined by underscores), or simple concatenation (COPY_IDENT). Spoken Java speakers may use the word "quote" in place of the quote character, so there are two more actions used to substitute the quote character for the word at the beginning and end of strings and character literals (COPY_CHAR_LEXEME, COPY_STRING_LEXEME).

Translation is done through a top-down depth-first traversal through the parse tree. The node type (grammar production) for each node in the parse tree is used to dispatch into the specification table. Each child of the node is processed as the specification dictates for the right-hand side of that production. If a sequence or optional node is found without special handling, the algorithm simply continues. If ambiguities are found, one of the alternatives is chosen to be translated, ignoring the others. To translate all ambiguous interpretations, the translation function must be called again on the same parse tree after the caller changes the order of the alternatives in the ambiguous node to allow another alternative to be chosen. The result of translation is a string in the target language.

There are several special cases in the translator. When translating a type cast or array reference operation from Spoken Java to Java, the production's symbols must be reordered to conform to Java. Likewise, when converting from Java to Spoken Java, type cast operations must be reordered. In addition, when method calls have no arguments, their argument lists are replaced by the keyword NOARGS which is what the study found that programmers are likely to say.

Chapter 4

Analyzing Ambiguities

Programming by voice, the novel form of user interface described in this dissertation, enables the user to edit, navigate, and dictate code using voice recognition software. It uses a combination of commands and program fragments, rather than full-blown natural language. Spoken input, however, contains many lexical ambiguities, such as homophones,[1] misrecognized, unpronounceable, and concatenated words. When the input is natural language, it can be disambiguated by a hidden Markov model provided by the speech recognition vendor. However, when the input is a computer program, these natural language disambiguation rules do not apply. It is as if one were to use German language rules to understand English text. Some words and sentence structures are similar, but most are completely different. Not only do the ambiguities affect the voice-based programmer's ability to introduce code, they also affect the ability of the voice-based programmer to use similar sounding words in different contexts.

Traditional programming language analyses do not handle ambiguity, because languages were designed specifically to be unambiguous, with very precise syntax and semantics. This mathematical precision of programming languages is both a curse and a blessing. It is a curse for verbal entry of programs because humans do not speak punctuation or capitalization, they drop and reorder words, and speak in homophones – all features of a program that must be precisely written down. Fortunately, however, the same precision that appears to hinder system understanding of spoken programs is also the solution. We can analyze the program being written to disambiguate what the user spoke and deduce the correct interpretation. This cannot be done with natural language because natural language syntactic and semantic analysis is still infeasible, and natural language semantics are far more ambiguous than those of any programming language.

[1] Homophones are words that sound alike but have different spellings.

Using program analysis techniques we have adapted for speech, we use the program context to help choose from among many possible interpretations for a sequence of words uttered by the user. We present an example here. A programmer wants to insert text at the ellipses in the following block of code:

```
String filetoload = null;
InputStream stream = getStream();
try {
   ...
} catch (IOException e) {
   e.printStackTrace();
}
```

She says "file to load equals stream dot read string."

Let us look at the interpretation of just the first three words "file to load," considering variable spelling and word concatenization. It is possible to spell "to" as "too," "two" or "2". "Load" can also be spelled "lode" or "lowed." And either the first two words or the second two words can be concatenated together to form "fileto," and "toload", or all three words can be concatenated together to spell "filetoload." This makes 48 possible interpretations of the words (12 spelling combinations times 4 word concatenizations) that must be considered by the lexical and syntactic analyses in our system.

This and many other lexical and syntactic ambiguities form what we call *input stream ambiguities*. These kinds of ambiguities also appear in embedded languages and legacy languages. Unfortunately, many widely-used programming language analysis generators, among them the popular and successful Flex lexer generator and Bison parser generator, fail to handle input stream ambiguities. We have developed Blender, a combined lexer and parser generator that enables designers to handle ambiguities in spoken input. Blender produces parsers for a new parsing algorithm that we have created called XGLR (or eXtended Generalized LR). XGLR is an extension to GLR (Generalized LR) and is one algorithm in a family of parsing algorithms designed for analyzing ambiguities.

The result of a traditional LR parse is a parse tree. The GLR family of parsers (of which XGLR is a member) produces a forest of parse trees, each tree representing one valid parse of the input. In the example given above, there are many many valid parses. Sixteen possible structures for the three words "file to load" are shown in Table 4.1.

The user's utterance is entered at a specific location in a Java program, and must make sense in that context. Our system uses knowledge of the Java programming language as well as

file to load	**file 2 load**	**file toload**	**filetoload**
1. file to load	9. file 2load	12. file toload	15. filetoload()
2. file(to, load)	10. file(2, load)	13. file(toload)	16. filetoload
3. file(to.load)	11. (file, 2, load)	14. file.toload	
4. file(to(load))			
5. file.to(load)			
6. file.to.load			
7. file to.load			
8. file.to load			

Table 4.1: Sixteen possible parses for three spoken words, "file to load."

contextual semantic information valid where the utterance was spoken to disambiguate the parse forest and filter out the invalid structures.

To continue with the example above, using the system's knowledge of the programming language, we can immediately rule out interpretations 1, 6, 7, 9, and 12 because in Java, two names are not allowed to be separated by a space. Next, after having analyzed the context around the cursor position, it can be determined what variable and method names are currently in scope, (i.e. visible to the line of code that the programmer is entering). If a name is not visible, it must be illegal, and therefore an incorrect interpretation. In our programmer's situation, there are no variables named "file," so interpretations 5, 8, 11 and 14 can be ruled out. Likewise, there are no methods (i.e. functions) named "file," so interpretations 2, 3, 4, 10 and 13 are incorrect. Finally, program analysis informs the system that there is no method named "filetoload," thus ruling out interpretation 15. The remaining interpretation is 16, which is the correct one. There is a variable "filetoload" where all three uttered words are concatenated together, and where the middle homophone is spelled "to," and the final homophone is spelled "load."

It is possible to develop a hand-coded semantic analysis for Java that will perform the disambiguation. However, a better solution that can be applied to many programming languages is to automate the process. While lexer and parser generators are well-known and commonly used in production compilers and program analysis, name resolution and type checking are not often automated. In this dissertation, we have extended and implemented a formalism called the Inheritance Graph (IG), which was originally described in the Ph.D. dissertation of Phillip Garrison [33]. The IG is a graph-based data structure which represents the names, scopes and bindings found in a program. Name-kind-type bindings flow along edges in the graph into nodes that represent program scopes, such as the one inside the `try` block above. When this flow process, known as propagation, finishes, each node in the graph will have a list of all bindings visible from that scope in the pro-

gram. These lists are suitable for answering the questions, “what does this name mean?” and “what names are visible from this point in the program?”. By looking up the interpretations of the user’s words produced by XGLR, we can figure out which words are legitimate and which words are not, and easily rule out semantically-invalid interpretations. Usually there will be just one interpretation left, but in case several cannot be ruled out, the programmer must eventually choose the correct one.

In the next two chapters, we describe two significant contributions in program analysis technology designed for ambiguities, XGLR (and its associated lexer and parser generator Blender), and the Inheritance Graph. The XGLR section appeared in published form in LDTA 2004 [8]. The Inheritance Graph is joint work with Johnathon Jamison, another graduate student in our research group.

Chapter 5

XGLR – An Algorithm for Ambiguity in Programming Languages

In the previous chapter, we motivated the creation of a new program analysis that can handle input stream ambiguities. In this chapter, we introduce our new XGLR analysis and discuss it in detail.

Many input stream ambiguities arise from speaking programming languages. Other forms of input stream ambiguities also exist. Legacy languages like PL/I and Fortran present difficulties to both a Flex-based lexer and an LALR(1) based parser. PL/I, in particular, does not have reserved keywords, meaning that `IF` and `THEN` may be both keywords and variables. A lexer cannot distinguish between those interpretations; only the parser and static semantics have enough context to choose among them. Fortran's optional whitespace rule leads to insidious lexical ambiguities. For example, `DO57I` can designate either a single identifier or `DO 57 I`, the initial portion of a Do loop. Without syntactic support, a particular character sequence could be interpreted using several sets of token boundaries. Feldman [30] summarizes other difficulties that arise in analyzing Fortran programs.

Embedded languages, in which fragments of one language can be embedded within another language, are in widespread use in common application domains such as Web servers (*e.g.* PHP embedded in XHTML), data retrieval engines (*e.g.* SQL embedded in C), and structured documentation (*e.g.* Javadoc embedded in Java). The boundaries between languages within a document can be either fuzzy or strict; detecting them might require lexical, syntactic, semantic or customized analysis.

The lack of composition mechanisms in Flex and Bison for describing embedded languages makes independent maintenance of each component language unwieldy and combined analysis awkward. Other language analyzer generators, such as ANTLR [74], ASF+SDF [50], or SPARK [5] provide better structuring mechanisms for language descriptions, but differing language conventions for comments, whitespace, and token boundaries complicate both the descriptions of embedded languages, and the analyses of their programs, particularly in the presence of errors. In developing analysis methods to handle spoken language, we also found new solutions for embedded languages.

Section 5.1 of this chapter summarizes the Harmonia framework within which our enhanced methods are implemented. The methods described in Section 5.2 handle four kinds of input streams: (1) single spelling; single lexical type, (2) multiple spellings; single lexical type, (3) single spelling; multiple lexical types, and (4) multiple spellings; multiple lexical types. The last three are ambiguous. Combinations of these ambiguities arise in different forms of embedded languages as well as in spoken languages. The handling of input streams containing such combinations is presented in Section 5.4. Some of these ambiguities have also been addressed in related work, which is summarized in Section 5.6.

1. **Single spelling; single lexical type.** This is normal, unambiguous lexing (*i.e.* a sequence of characters produces a unique sequence of tokens). We illustrate this case to show how lexing and parsing work in the Harmonia analysis framework.

2. **Multiple spellings; single lexical type.** Programming by voice introduces potential ambiguities into programming that do not occur when legal programs are typed. If the user speaks a homophone which corresponds to multiple lexemes (for example, `i` and `eye`), and all the lexemes are of the same lexical type (the token `IDENTIFIER`), using one or the other homophone may change the meaning of the program. Multiple spellings of a single lexical type might also be used to model voice recognition errors or lexical misspellings of typed lexemes (*e.g.* the identifier `counter` occurring instead as `conter`).

3. **Single spelling; multiple lexical types.** Most languages are easily described by separating lexemes into separate categories, such as keywords and identifiers. However, in some languages, the distinction is not enforced by the language definition. For instance, in PL/I, keywords are not reserved, leading a simple lexeme like `IF` or `THEN` to be interpreted as both a keyword and an identifier. In such cases, a single character stream is interpreted by a lexer

as a unique sequence of lexemes, but some lexemes may denote multiple alternate tokens, which each have a unique lexical type.

4. **Multiple spellings; multiple lexical types.** Sometimes a user might speak a homophone (*e.g.*, "for", "4" and "fore") that not only has more than one spelling, but whose spellings have distinct lexical types (*e.g.* keyword, number and identifier).

5. **Embedded languages.** Two issues arise in the analysis of embedded languages – identifying the boundaries between languages, and analyzing the outer and inner languages according to their differing lexical, structural, and semantic rules. Once the boundaries are identified, any ambiguities in the inner and outer languages can be handled as if embedding were absent. However, ambiguity in identifying a boundary leads to ambiguity in which language's rules to apply when analyzing subsequent input. Virtually all programming languages admit simple embeddings, notably strings and comments. The embedding in an example such as Javadoc within Java is more complex. These embeddings are typically processed by *ad hoc* techniques. When properly described, they can be identified in a more principled fashion. For example, Synytskyy, Cordy, and Dean [94] use island grammars to analyze multilingual documents from web applications. Their approach is summarized in Section 5.6.

The results described in this chapter require modifications to conventional lexers and parsers, whether batch or the incremental versions used in interactive environments. Our approach is based on GLR parsing [95], a form of general context free parsing based on LR parsing, in which multiple parses are constructed simultaneously. Even without input ambiguities, the use of GLR instead of LR parsing enables support for ambiguities during the *analysis* of an input stream. GLR tolerates local ambiguities by forking multiple parses, yet is efficient because the common parts of the parses are shared. In addition, for the syntax specifications of most programming languages, the amount of ambiguity that arises is bounded and fairly small. Our contribution is to generalize this notion of ambiguity, and the GLR parsing method, to parse inputs that are locally different (whether due to the embedding of languages, the presence of homophones or other lexically-identified ambiguities). We call this enhanced parser XGLR.

We have strengthened the language analysis capabilities of our Harmonia analysis framework [11, 36] to handle these kinds of ambiguities. Our research in programming by voice requires interactive analysis of input stream ambiguities. Harmonia can now identify ambiguous lexemes in spoken input. In addition, Harmonia's new ability to embed multiple formal language descrip-

tions enables us to create a voice-based command language for editing and navigating source code. This new input language combines a command language written in a structured, natural-language style (with a formally specified syntax and semantics) with code excerpts from the programming language in which the programmer is coding.

To realize these additional capabilities, the parser requires additional data structures to maintain extra lexical information (such as a lookahead token and a lexer state for each parse), as well as an enhanced interface to the lexer. These changes enable the XGLR parser to resolve *shift–shift* conflicts that arise from the ambiguous nature of the parser's input stream. The lexer must be augmented with a bit of extra control logic. A completely new lexer and parser generator called Blender was developed. Blender produces a lexical analyzer, parse tables and syntax tree node C++ classes for representing syntax tree nodes in the parse tree. It enables language designers to easily describe many classes of embedded languages (including recursively nested languages), and supports many kinds of lexical, structural and semantic ambiguities at each stage of analysis. In the next section, we summarize the structure of incremental lexing and GLR parsing, as realized in Harmonia. The changes to support input ambiguity and the design of Blender follow.

5.1 Lexing and Parsing in Harmonia

Harmonia is an open, extensible framework for constructing interactive language-aware programming tools. Programs can be edited and transformed according to their structural and semantic properties. High-level transformation operations can be created and maintained in the program representation. Harmonia furnishes the XEmacs [109] and Eclipse [23] programming editors with interactive, on-line services to be used by the end user during program composition, editing and navigation.

Support for each user language is provided by a plug-in module consisting of a lexical description, syntax description and semantic analysis definition. The framework maintains a versioned, annotated parse tree that retains all edits made by the user (or other tools) and all analyses that have ever been executed [105]. When the user makes a keyboard-based edit, the editor finds the lexemes (i.e., the terminal nodes of the tree) that have been modified and updates their text, temporarily invalidating the tree because the changes are unanalyzed. If the input was spoken, the words from the voice recognizer are turned into a new unanalyzed terminal node and added to the appropriate location in the parse tree. These changes make up the most recently edited version (also called the last edited version). This version of the tree and the pre-edited version are used by an

incremental lexer and parser to analyze and reconcile the changes in the tree.

Harmonia employs incremental versions of lexing and sentential-form GLR parsing [104, 106, 107, 108] in order to maintain good interactive performance. For those unfamiliar with GLR, one can think of GLR parsing as a variant of LR parsing. In LR parsing, a parser generator produces a parse table that maps a parse state/lookahead token pair to an action of the parser automaton: shift, reduce using a particular grammar rule, or declare error. The table contains only one action for each parse state/lookahead pair. Multiple potential actions (conflicts) must be resolved at table construction time. In addition to the parse table and the driver, an LR parser consists of an input stream of tokens and a stack upon which to shift grammar terminals and nonterminals. At each step, the current lookahead token is paired with the current parse state and looked up in the parse table. The table tells the parser which action to perform and, in the absence of an error, the parse state to which it should transition.

The GLR algorithm used in Harmonia is similar to that described by Rekers [84] and by Visser [100]. In GLR parsing, conflict resolution is deferred to runtime, and all actions are placed in the table. When more than one action per lookup is encountered, the GLR parser forks into multiple parsers sharing the same automaton, the same initial portion of the stack, and the same current state. Each forked parser performs one of the actions. The parsers execute in pseudo parallel, each executing all possible parsing steps for the next input token before the input is advanced (and forking additional parsers if necessary), and each maintaining its own additional stack. When a parser fails to find any actions in its table lookup, it is terminated; when all parsers fail to make progress, the parse has failed, and error recovery ensues. Parsers are merged when they reach identical states after a reduce or shift action. Thus conceptually, the forked parsers either construct multiple subtrees below a common subtree root, representing alternative analyses of a portion of the common input, or they eventually eliminate all but one of the alternatives.

The basic non-incremental form of the GLR algorithm (before any of our changes) is shown in Figure 5.1.[1] In GLR parsing, each parser stack is represented as a linked structure so that common portions can be shared. Each parser state in a list of parsers contains not only the current state recorded in the top entry, but also pointers to the rest of all stacks for which it is the topmost element. In Figures 5.1, 5.2, and 5.3, the algorithm is abstracted to show only those aspects changed by our methods. In particular, parse stack sharing is implicit. Thus *push* q *on stack* p means to advance all the specified parsers with current state p to current state q. The current lookahead

[1] The addition of incrementality is not essential to understanding the changes made here and is not shown.

```
GLR-PARSE()
init active-parsers list to parse state 0
init parsers-ready-to-act list to empty
while not done
   PARSE-NEXT-SYMBOL()
   if accept before end of input
      invoke error recovery
accept
```

```
PARSE-NEXT-SYMBOL()
lex one lookahead token
init shiftable-parse-states list to empty
copy active-parsers list to parsers-ready-to-act list
while parsers-ready-to-act list ≠ ∅
   remove parse state p from list
   DO-ACTIONS(p)
SHIFT-A-SYMBOL()
```

Figure 5.1: A non-incremental version of the unmodified GLR parsing algorithm. Continued in Figures 5.2 and 5.3.

token is held in a global variable *lookahead*.

In a batch LR or GLR parse, the sentential form associated with a parser at any stage is the sequence of symbols on its stack (read bottom-to-top) followed by the sequence of remaining input tokens. Conceptually, they represent a parse forest that is being built into a single parse tree. In an incremental parser, both the symbols on the stack and the symbols in the input may be parse (sub)trees (see Figure 5.4) – one can think of them as potentially a non-canonical sentential form. The goal of an incremental or change-based analysis is to preserve as much as possible of the parse prior to a change, updating it only as much as is needed to incorporate the change.

The result of lexing and parsing is sometimes a parse forest made up of all possible parse trees. Semantic analysis must be used to disambiguate any valid parses that are incorrect with respect to the language semantics. For example, to disambiguate identifiers that ought to be concatenated (but were entered as separate words because they came from a voice recognizer) the semantic phase can use symbol table information to identify all in-scope names of the appropriate kind (method name, field name, local variable name, etc.) that match a concatenated sequence of identifiers that is semantically correct as shown in the example at the beginning of Chapter 4.

```
DO-ACTIONS(parse state p)
look up actions[p×lookahead]
for each action
  if action is SHIFT to state x
    add <p, x> to shiftable-parse-states
  if action is REDUCE by rule y
    if rule y is accepting reduction
      if at end of input return
      if parsers-ready-to-act list = ∅
        invoke error recovery
      return
    DO-REDUCTIONS(p, rule y)
    if no parsers ready to act or shift
      invoke error recovery and return
  if action is ERROR and no parsers ready to act or shift
    invoke error recovery and return

SHIFT-A-SYMBOL()
clear active-parsers list
for each <p, x> ∈ shiftable-parse-states
  if parse state x ∈ active-parsers list
    push x on stack p
  else
    create new parse state x
    push x on stack p
    add x to active-parsers list
```

Figure 5.2: A non-incremental version of the unmodified GLR parsing algorithm. Continued in Figure 5.3.

Care with analysis must be taken if an inner language can access the semantics of the outer (*e.g.* Javascript can reference objects from the HTML code in which it is embedded). Semantic analyses techniques are discussed in Chapter 6.

5.2 Ambiguous Lexemes and Tokens

In the introduction to this chapter we classified token ambiguities into four types (including unambiguous tokens). We next explain how these situations are handled.

```
DO-REDUCTIONS(parse state p, rule y)
for each parse state p⁻ below RHS(rule y) on a stack for parse state p
  let q = GOTO state for actions[p⁻ × LHS(rule y)]
  if parse state q ∈ active-parsers list
    if p⁻ is not immediately below stack for parse state q
      push q on stack p⁻
      for each parse state r such that
            r ∈ active-parsers list and r ∉ parsers-ready-to-act list
          DO-LIMITED-REDUCTIONS(r)
  else
    create new parse state q
    push q on stack p⁻
    add q to active-parsers list
    add q to parsers-ready-to-act list


DO-LIMITED-REDUCTIONS(parse state r)
look up actions[r × lookahead]
for each REDUCE by rule y action
  if rule y is not accepting reduction
    DO-REDUCTIONS(r, rule y)
```

Figure 5.3: The third portion of a non-incremental version of the unmodified GLR parsing algorithm.

5.2.1 Single Spelling – One Lexical Type

Unambiguous lexing and parsing is the normal state of our analysis framework. Programming languages have mostly straightforward language descriptions, only incorporating bounded ambiguities when described using GLR. Thus, the typical process of the lexer and parser is as follows. The incremental parser identifies the location of the edited node in the last edited parse tree and invokes the incremental lexer. The incremental lexer looks at a previously computed *lookback* value (stored in each token) to identify how many tokens back in the input stream to start lexing due to the change in this token.[2] The characters of the starting token are fed to the Flex-based lexical analyzer one at a time until a regular expression is matched. The action associated with the regular expression creates a single, unambiguous token, which is returned to the parser to use as its lookahead symbol. In response to the parser asking for tokens, lexing continues until the next token is

[2]Lookback is computed as a function of the number of lookahead characters used by the batch lexer when the token is lexed. [104]

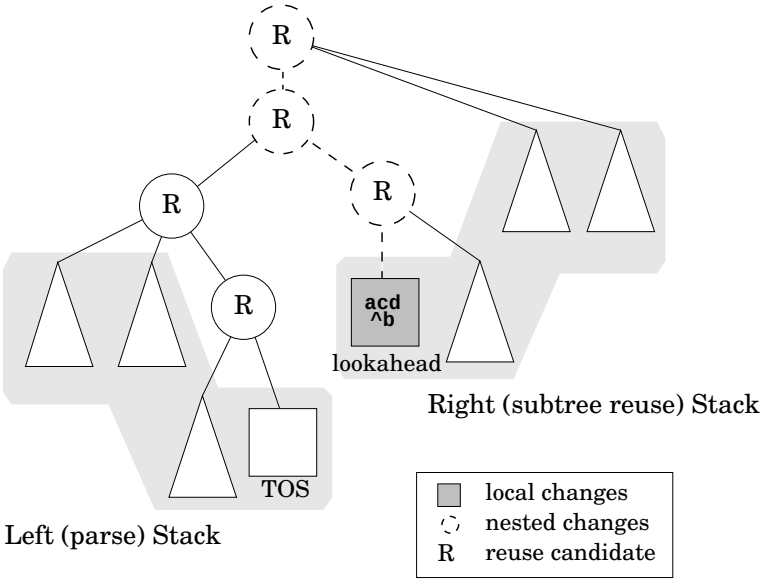


Figure 5.4: A change in the spelling of an identifier has resulted in a split of the parse tree from the root to the token containing the modified text. In an incremental parse, the shaded portion on the left becomes the initial contents of the parse stack. The shaded portion on the right represents the potentially reusable portion of the input stream. Parsing proceeds from the TOS (top of stack) until the rest of the tree in the input stream has been reincorporated into the parse. This figure originally appeared in Wagner's dissertation [104].

a token that is already in the edited version of the syntax tree. (The details of parser incrementality are not essential to this discussion and are omitted for brevity. Notice that additional information must be stored in each tree node to support incrementality.)

5.2.2 Single spelling – Multiple Lexical Types

If a single character sequence can designate multiple lexical types, as in PL/I, tokens are created for each interpretation (containing the same text, but differing lexical types) and are all inserted into an *AmbigNode* container. When the lexer/parser interface sees an AmbigNode, namely, multiple alternate tokens, that AmbigNode represents a shift–shift conflict for the parser. A new lexer instance is created for each token, and a separate parser is created for each lexer instance. Thus each parser has its own (possibly shared) lexer and its own lookahead token. The GLR parse is carried out as usual, except that instead of a global lookahead token, the parsers have local lookaheads with a shared representation. Due to this change, the criteria for merging parsers includes not only that the parse states are equal, but that the lookahead token and the state of each parser's lexer

```
PARSE-NEXT-SYMBOL()
for each parse state p ∈ active-parsers list
  set lookahead_p to first token lexed by lex_p
  if lookahead_p is ambiguous
    let each of q_1 .. q_n = copy parse state p
    for each parse state q ∈ q_1 .. q_n
      for each alternative a from lookahead_p
        set lookahead_q to a
      add q to active-parsers list
init shiftable-parse-states list to empty
copy active-parsers list to parsers-ready-to-act list
while parsers-ready-to-act list ≠ ∅
  remove parse state p from list
  DO-ACTIONS(p)
SHIFT-A-SYMBOL()
```

Figure 5.5: Part of the XGLR parsing algorithm modified to support ambiguous lexemes.

instance are the same as well.

In Figure 5.5 is our modification of the *PARSE-NEXT-SYMBOL()* function. Note that both *lex* and *lookahead* are now associated with a parser *p* rather than being global. Not shown are the changes to the parser merging criteria in *DO-REDUCTIONS()* and to the creation of new parse states (which should be associated with the current *lex* and *lookahead*). In addition, each lookup must reference the lookahead associated with its parser – for example, $actions[p \times lookahead_p]$

5.2.3 Multiple Spellings – One Lexical Type

Harmonia's voice-based editing system looks up words entered by voice recognition in a homophone database to retrieve all possible spellings for that word. The lexer is invoked on each word to discover its lexical type and create a token to contain it. If all alternatives have the same lexical type (*e.g.* all are identifiers), they are returned to the parser in a container token called a *MultiText*, which to the parser appears as a single, unambiguous token of a single lexical type. Once incorporated into the parse tree, semantic analysis can be used to select among the homophones.

A similar mechanism could be used for automated semantic error recovery. Identifiers can easily be misspelled by a user when typing on a keyboard. Compilers have long supported substituting similarly spelled (or phonetically similar) words for the incorrect identifier. In an incremental

setting, where the program, parse, and symbol table information are persistent, error recovery could replace the user's erroneous identifier with an ambiguous variant that contains the original identifier along with possible alternate spellings. Further analysis might be able to automatically choose the proper alternative based on the active symbol table. We have not yet investigated this application.

5.2.4 Multiple Spellings – Multiple Lexical Types

If the alternate spellings for a spoken word (as described above) have differing lexical types (such as `4`/`for`/`fore`), they are returned to the parser as individual tokens grouped in the same AmbigNode container described above. When the lexer/parser interface sees an AmbigNode, it forks the parser and lexer instance, and assigns one token to each lexer instance.[3] The state of each lexer instance must be reset to the lexical state encountered after lexing its assigned alternative, since each spelling variant may traverse a different path through the lexer automaton.[4] Once each token is re-lexed, it is returned to its associated parser to be used as its lookahead token and shifted into the parse tree.

5.3 Lexing and Parsing with Input Stream Ambiguities

The input stream ambiguities described in the previous section require several changes to the GLR algorithm. We illustrate the new algorithm in Figures 5.6, 5.7, 5.8, 5.9, and 5.10. Lines that have been altered or added from the original GLR algorithm are indicated with boxes.

When there are lexical ambiguities (multiple lexical types) in the input stream, a new parser must be forked for each interpretation of an ambiguous token. This forking occurs in *SETUP-LOOKAHEADS()*. The ambiguous lookahead tokens that caused the parsers to fork are joined into an equivalence class for later use during parser merging (explained below). After shifting symbols, parser merging may cause multiple parsers incorrectly to share a lexer. The function of *SETUP-LEXER-STATES()* is to ensure that each parser's lexer instance is unique.

Next, if each parser has its own private lexer instance, and each lexer instance is in a different lexical state when reading the input stream, then the input streams may diverge at their

[3] Note that the main characteristic distinguishing AmbigNodes from MultiTexts is that AmbigNodes have multiple lexical types where MultiTexts have only one. Since all spellings of a MultiText have the same lexical type, the parser need not (in fact, must not) fork when it sees one. The parser only forks when the aggregate token it receives contains multiple lexical types that could cause the forked parsers to take different actions.

[4] Note that we do not reset the lexical state on a single spelling – multiple lexical type ambiguity because the text of each alternative (and thus the lexer's path through its automaton) is the same, ending up in the same lexical state.

XGLR-PARSE()
init *active-parsers list to parse state 0*
init *parsers-ready-to-act list to empty*

init *parsers-at-end list to empty*
init *lookahead-to-parse-state map to empty*
init *lookahead-to-shiftable-parse-states map to empty*
while *active-parsers list* $\neq \emptyset$
  PARSE-NEXT-SYMBOL(false)
copy *parsers-at-end list to active-parsers list*
clear *parsers-at-end list*
PARSE-NEXT-SYMBOL(true)

accept

SETUP-LEXER-STATES()
for each *pair of parse states p, q* $\in$ *active-parsers list*
  if *lexer state of* lex_p = *lexer state of* lex_q
    set lex_p *to* **copy** lex_q

Figure 5.6: A non-incremental version of the fully modified XGLR parsing algorithm. The portions of the algorithm contained within the boxes is changed from the original GLR algorithm. Continued in Figure 5.7.

token boundaries, with some streams producing fewer tokens and some producing more. This may cause a given parser to be at a different position in the input stream from the others, which is a departure from the traditional GLR parsing algorithm in which all parsers are kept in sync shifting the same lookahead token during each major iteration. Unless we are careful, this could have serious repercussions on the ability of parsers to merge, as well as performance implications if one parser were forced to repeat the work of another.

To solve this problem, we observe that any two parsers that have forked will only be able to merge once their parse state, lexer state and lookahead tokens are equivalent.[5] For out-of-sync parsers, this can only happen when the input streams converge again after the language boundary ambiguities have been resolved. However, in the XGLR algorithm given in Figure 5.1, only the *active-parsers* list is searched for mergeable parsers. If a parser p is more than one input token ahead of a parser q, q will no longer be in the *active-parsers* list when p will be ready to merge with it. If the merge fails to occur, parser p may end up repeating the work of parser q.

We introduce a new data structure, a map from a lookahead token to the parsers with

[5] At the end of the input stream when there is no more input to lex, it is not important to check for lexer state equality.

```
PARSE-NEXT-SYMBOL(bool finish-up?)
SETUP-LEXER-STATES()
SETUP-LOOKAHEADS()
if not finish-up?
   FILTER-FINISHED-PARSERS()
   if active-parsers list is empty? return
init shiftable-parse-states list to empty
copy active-parsers list to parsers-ready-to-act list
while parsers-ready-to-act list ≠ ∅
   remove parse state p from list
   DO-ACTIONS(p)
SHIFT-A-SYMBOL()
```

```
SETUP-LOOKAHEADS()
for each parse state
      p ∈ active-parsers list
   set lookahead_p to first token lexed by lex_p
   add <(offset of lookahead_p)×lookahead_p> to offset-to-lookaheads map
   if lookahead_p is ambiguous
      let each of q_1 .. q_n = copy parse state p
      for each parse state q ∈ q_1 .. q_n
         for each alternative a from lookahead_p
            set lookahead_q to a
            add lookahead_q to equivalence class for a
         add q to active-parsers list
for each parse state p ∈ active-parsers list
   add <lookahead_p×p> to lookahead-to-parse-state map
```

Figure 5.7: The second portion of a non-incremental version of the fully modified XGLR parsing algorithm. The portions of the algorithm contained within the boxes are changed from the original GLR algorithm. Continued in Figure 5.8.

that lookahead. The map is initialized to empty in *XGLR-PARSE()*, and is filled with each parser in the *active-parsers* list after each lookahead has been lexed in *PARSE-NEXT-SYMBOL()*. Any new parsers created during *DO-REDUCTIONS()* are added to the map. In *DO-REDUCTIONS()*, when a parser searches for another to merge with, instead of searching the *active-parsers* list, it searches the list of parsers in the range of the map associated with the parser's lookahead. In the case where all parsers remained synchronized at the same lookahead terminal, this degenerates to the old behavior. But when parsers get out of sync, it enables the late parser to merge with a parser that has already moved past the terminal, thereby avoiding repeated work.

```
DO-ACTIONS(parse state p)
look up actions[p×lookahead_p]
for each action
  if action is SHIFT to state x
    add <p, x> to shiftable-parse-states
    add <lookahead_p×p> to lookahead-to-shiftable-parse-states map
  if action is REDUCE by rule y
    if rule y is accepting reduction
      if lookahead_p is end of input
        return
      if no parsers ready to act or shift or at end of input
        invoke error recovery
      return
    DO-REDUCTIONS(p, rule y)
    if no parsers ready to act or shift
      invoke error recovery and return
  if action is ERROR and no parsers
      ready to act or shift or at end of input
    invoke error recovery and return

FILTER-FINISHED-PARSERS()
for each parse state p ∈ active-parsers list
  if lookahead_p = end of input?
    remove p from active-parsers list
    add p to parsers-at-end list
```

Figure 5.8: The third portion of a non-incremental version of the fully modified XGLR parsing algorithm. The portions of the algorithm contained within the boxes are changed from the original GLR algorithm. Continued in Figure 5.9.

Parser merging in XGLR contains one more potential pitfall that must be addressed in the implementation of the algorithm. The criteria for parser merging compares two lookahead tokens for equivalence. Usually, equivalence is an equality test, but for tokens that caused the parsers to fork, the algorithm tests each token for membership in the same equivalence class (assigned in *SETUP-LEXER-LOOKAHEADS()*). We use this equivalence to properly merge the parse trees formed by the reduction of each parser in *DO-REDUCTIONS*. Normally, both parsers involved in successful merge would share a p^{-} during the reduce action. Parsers that were created by forking at an input stream ambiguity do not because the parser fork occurred *before* the shift of the equivalent tokens, not after. Even though all the conditions for parser merging are met, the implementation of

SHIFT-A-SYMBOL()
clear *active-parsers list*
for each $<p, x> \in$ *shiftable-parse-states*
if *p is not an accepting parser*
if *parse state* $x \in$ *active-parsers list*
push *x on stack p*
else
create *new parse state x with* $lookahead_p$ *and copy of* lex_p
push *x on stack p*
add *x to active-parsers list*

DO-REDUCTIONS(parse state p, rule y)
for each *equivalent parse state* p^- *below RHS(rule y) on a stack for parse state p*
let *q = GOTO state for* $actions[p^- \times LHS(rule\ y)]$
if *parse state* $q \in lookahead\text{-}to\text{-}parse\text{-}state[lookahead_p]$ *and* $lookahead_q \cong lookahead_p$
and ($lookahead_p$ *is end of input or lexer state of* lex_q *= lexer state of* lex_p)
if p^- *is not immediately below q on stack for parse state q*
push *q on stack* p^-
for each *parse state r such that* $r \in$ *active-parsers list and* $r \notin$ *parsers-ready-to-act list*
DO-LIMITED-REDUCTIONS(r)
else
create *new parse state q with* $lookahead_p$ *and copy of* lex_p
push *q on stack* p^-
add *q to active-parsers list*
add *q to parsers-ready-to-act list*
add $<lookahead_q \times q>$ *to lookahead-to-parse-state map*

Figure 5.9: The fourth portion of a non-incremental version of the fully modified XGLR parsing algorithm. The portions of the algorithm contained within the boxes are changed from the original GLR algorithm. Continued in Figure 5.10.

the algorithm must ensure an equivalence between all possible parsers p^- that could shift any of the lookahead tokens in the equivalence class. We use a map to record all parsers that can immediately shift a particular lookahead token (the *lookahead-to-shiftable-parse-states* map). The set of all *equivalent* parsers p^- is the range of the *lookahead-to-shiftable-parse-states* map with the domain being all lookahead tokens in the equivalence class of token $lookahead_p$.

Since any parser may be out of sync with other parsers, the end of the input stream may be reached by some parsers before others. These parsers are stored separately in the *parsers-at-end* list because it simplifies the control flow logic of the algorithm to have all parsers that are

```
DO-LIMITED-REDUCTIONS(parse state r)
look up actions[r × lookahead_r]
for each REDUCE by rule y action
  if rule y is not accepting reduction
    DO-REDUCTIONS(r, rule y)
```

Figure 5.10: The remainder of a non-incremental version of the fully modified XGLR parsing algorithm. The portions of the algorithm contained within the boxes are changed from the original GLR algorithm.

ready to accept the input accept in the same call to *PARSE-NEXT-SYMBOL()*. We add a Boolean argument *finish-up?* to *PARSE-NEXT-SYMBOL()* to indicate this final invocation and we call the *FILTER-FINISHED-PARSERS()* function to move the finished parsers to the *parsers-at-end* list.

In practice, XGLR uses more memory than GLR. In addition to the two maps above, which cannot be pruned during the parse (reductions may require looking up any already parsed token in the map), the lack of synchronization of parsers requires each parser to hold extra state that is global in GLR. This memory requirement grows linearly as the number of parsers, or equivalently, as the number of dynamic ambiguities in the program discovered during the parse.

5.4 Embedded Languages

In addition to using this algorithm for programming by voice in this dissertation, it is also used to write language descriptions for embedded languages and language dialects. Using Blender, the outer and inner languages that constitute an embedded language can be specified by two completely independent language definitions, for example, one for PHP and one for HTML, which are composed to produce the final language analysis tool. Language dialects contain related language definitions, where one is an extension of the other. For example, Titanium, a parallel programming language [110], is a dialect and superset of Java 1.4. We can describe Titanium using two grammars and two lexical descriptions. The outer grammar and lexical description is for Java 1.4; the inner language consists of extra (and altered) grammar productions as well as new lexical rules for Titanium's new keywords.

Embedded and dialect language descriptions may be arbitrarily nested and mutually recursive. It is the job of the language description writer to provide appropriate boundary descriptions.

5.4.1 Boundary Identification

In embedded languages, boundaries between languages may be designated by context (*e.g.*, the format control in C's `printf` utility), or by delimiter tokens before and after the inner language occurrence. The delimiters may or may not be distinct from one another; they may or may not belong to the outer (resp. inner) language, and they may or may not have other meanings in the inner (resp. outer) language. We refer to these delimiters as a *left boundary token* and a *right boundary token*. Older legacy languages, usually those analyzed by hand-written lexers and parsers, tend to have more fuzzy boundaries where either one of these boundary tokens may be absent or confused for whitespace. For example, in the description format used by Flex, the boundary between a regular expression and a C-based action in its lexical rules is simply a single character of whitespace followed by an optional left curly brace.

One technique for identifying boundaries is to use a special program editor that understands the boundary tokens that divide the two languages (*e.g.*, PHP embedded in XHTML) and enforces a high-level document/subdocument editing structure. The boundary tokens are fixed, and once inserted, can not be edited or removed without removing the entire subdocument. The two languages can then be analyzed independently.

Another technique is to use regular expression matching or a simple lexer to identify the boundary tokens in the document and use them as an indication to switch analysis services to or from the inner language. These services are usually limited to lexically based ones, such as syntax highlighting or imprecise indentation. More complex services based on syntax analysis cannot easily be used, since the regular expressions are not powerful enough to determine the boundary tokens accurately. In some cases, it might be possible to use a coarse parse such as Koppler's [55], but we have not explored that alternative.

Some newer embedded languages maintain lexically identifiable boundaries (*e.g.* PHP's starting token is `<?php` and its ending token is `?>`). Others contain boundaries that are only structurally or semantically detectable (*e.g.* Javascript's left boundary is `<script language = javascript>`).

5.4.2 Lexically Embedded Languages

Lexically embedded languages are those where the inner language has little or no structure and can be analyzed by a finite automaton. To give an example, the typical lexical description for the Java language includes standard regular expressions for keywords, punctuation, and identifiers. The

most complicated regular expressions are reserved for strings and comments. A string is a sequence of characters bounded by double quote characters on either side. A comment is a sequence of characters bounded by a `/*` on the left and a `*/` on the right. Inside these boundary tokens, the traditional rules for Java lexing are suspended — no keywords, punctuation or identifiers are found within. Most description writers will "turn off" the normal Java lexical rules upon seeing the left boundary token, either by using lexer "condition" states,[6] or by storing the state in a global variable. When the right boundary token is detected, the state is changed back to the initial lexer state to begin detecting keywords again.

From the perspective of an embedded language, it is obvious that strings and comments form inner languages within the Java language that use completely different lexical rules. Using Harmonia, we can split these out into separate components and thereby clean up the Java lexical specification.

In the case of a string within a Java program, the two boundary tokens are identical, and lexically identifiable by a simple regular expression. However, aside from a rule that double quote may not appear unescaped inside a string, the double quotes that form the boundaries are not part of the string data. This is also true for comments — the boundary tokens identify the comment to the parser, but do not make up the comment data.

5.4.3 Syntactically Embedded Languages

Syntactically embedded languages are those where the inner language has its own grammatical structure and semantic rules. Compilers for syntactically embedded languages typically use a number of *ad hoc* techniques to process them. One common technique is to ignore the inner language, for example, as is done with SQL embedded in PHP. PHP analysis tools know nothing about the lexical or grammatical structure of SQL, and in fact, treat the SQL code as a string, performing no static checking of its correctness.[7] Similarly, in Flex, C code is passed along as unanalyzed text by the Flex analyzer, and subsequently packaged into a C program compiled by a conventional C compiler. The lack of static analysis leaves the programmer at risk for runtime errors that could have been caught at compile time.

It is sometimes possible to analyze the embedded program. The embedded program can

[6] Condition states are explicitly declared automaton states in Flex-based lexical descriptions. They are often used to switch sub-languages.

[7] This incomplete and inappropriate lexing forces programmers to escape characters in their embedded SQL queries that would not be necessary when using SQL alone.

be segmented out and analyzed as a whole program or a program fragment independent of the outer program. This technique will not work, however, if the embedded program refers to structures in the outer program, or vice versa. In addition, the embedded program may not be in complete form. For example, it may be pieced together from distinct strings or syntactic parts by the execution of the outer program. Gould, Su and Devanbu [34] describe a static analysis of dynamically-generated SQL queries embedded in Java programs that can identify some potential errors. In general, to analyze a particular embedding of one language in another, a special purpose analysis is required, and often may not exist.

In the next section, we show how language descriptions are written in Blender, our combined lexer and parser generator tool.

5.4.4 Language Descriptions for Embedded Languages

Lexical descriptions are written in a variant of the format used by Flex. The header contains a set of token declarations which are used to name the tokens that will be returned by the actions in this description. At the beginning of a rule is a regular expression (optionally preceded by a lexical condition state) that when matched creates a token of the desired type(s) and returns it to the parser.

Grammar descriptions are written in a variant of the Bison format. Each grammar consists of a header containing precedence and associativity declarations, followed by a set of grammar productions. One or more `%import-token` declarations are written to specify which lexical descriptions to load (one of which is specified as the default) in order to find tokens to use in this grammar. In addition to importing tokens, a grammar may import nonterminals from another grammar using the `%import-grammar` declaration. Grammar productions do not have user-described actions.[8] The only action of the runtime parser is to produce a parse tree/forest from the input. The language designer writes a tree-traversing semantic analysis phase to express any desired actions.

Imported (non-default) terminals and nonterminals are referred to in this paper as `symbol`$_{language}$. An imported symbol causes an inner language to be embedded in the outer language.

[8] Because there are multiple parses with differing semantics, some of which may fail, it is tricky to get those actions right for GLR parsing, as discussed by McPeak [67].

5.4.5 Lexically Embedded Example

An example of a comment embedded in a Java program is:

```
/* Just a comment */
```

To embed the comment language in the outer Java grammar, the following rule might be added:

COMMENT → SLASHSTAR COMMENTDATA*comment-lang* STARSLASH

In Blender, boundary tokens for an inner language are specified with the outer language, so that the outer analyzer can detect the boundaries. The data for the inner language is written in a different specification, named comment-lang in the example, which is imported into the Java grammar. In this simple case, the embedding is lexical. Comment boundary tokens are described by regular expressions that detect the tokens `/*` and `*/`. They are placed in the main Java lexical description (the one that describes keywords, identifiers and literals).

The comment data can be described by the following Flex lexical rule, which matches all characters in the input including the carriage returns.

```
.|[\r\n]     { yymore(); break; }
```

However, this specification would read beyond the comment's right boundary token. Our solution, which is specialized to the peculiarities of a Flex-based lexer (and might be different in a different lexer generator), is to introduce a special keyword, `END_LEX`, into any lexical description that is intended to be embedded in an outer language. `END_LEX` will stand in for the regular expression that will detect the `*/`. Blender will automatically insert this regular expression based on the right boundary token following the `COMMENTDATA` terminal. For those familiar with Flex, the finalized description would look like:

```
%{  int comment_length;  %}
%token COMMENTDATA
%%
END_LEX   { yyless(comment_length); RETURN_TOKEN(COMMENTDATA); }
.|[\r\n]  { yymore(); comment_length = yyleng; break;          }
```

We must be careful to insert this new `END_LEX` rule before the other regular expression due to Flex's rule precedence property (lexemes matching multiple regular expressions are associated with the first one), or Flex will miss the right boundary token. Also, since the `COMMENTDATA` lexeme will only be returned once the right boundary token has been seen, its text would accidentally include the boundary token's characters. We use Flex's `yyless()` construct to push the right

boundary token's characters back onto the input stream, making it available to be matched by a lexer for the outer language, and then return the COMMENTDATA lexeme.

This sort of lexical embedding enables one to reuse common language components in several programming languages. For example, even though Smalltalk and Java use different boundary tokens for strings (Java uses " and Smalltalk uses '), their strings have the same lexical content. Lexically embedding a language (such as this String language) enables a language designer to reuse lexical rules that may have been fairly complex to create, and might suffer from maintenance problems if they were duplicated.

Syntactically Embedded Example

Syntactic embedding is easier to perform because of the greater expressive power of context-free grammars. One simply uses nonterminals from the inner language in the outer language. Following is an example of a grammar for Flex lexical rules:

RULE → REGEXP_ROOT$_{regexp}$ WSPC CCODE

CCODE → LBRACE COMPOUND_STMT$_{C}$ RBRACE NEWLINE

| COMPOUND_STMT_NO_CR$_{C}$ NEWLINE

A Flex rule consists of a regular expression followed by an optionally-braced C compound statement. The regular expression is denoted by the REGEXP_ROOT nonterminal from the regexp grammar. The symbol WSPC denotes a white-space character. The compound statement is denoted by the COMPOUND_STMT from the C grammar. COMPOUND_STMT_NO_CR is the same nonterminal as COMPOUND_STMT but has been modified to disallow carriage returns as whitespace inside, as specified by the Flex manual.

We can now show one of the lexical ambiguities associated with legacy embedded languages. A left brace token is described by the character { in both Flex *and* in C. A compound statement in C may or may not be bracketed by a set of curly braces. When a left brace is seen, it can belong either to the outer language for Flex or to the inner C language. Choosing the right language usually requires contextual information that is only available to a parser. Even the parser can only choose properly when presented with both choices, a Flex left brace token and a C left brace token. This is another example of a single lexeme with multiple lexical types; its resolution requires enhancements to both the lexer and parser generators as well as enhancements to the parser.

In the next section, we show how embedded terminals and nonterminals are incorporated in our tools.

5.4.6 Blender Lexer and Parser Table Generation for Embedded Languages

When a Blender language description incorporates grammars for more than one language, the grammars are merged.[9] Each grammar symbol is tagged with its language name to ensure its uniqueness. Blender then builds an LALR(1) parse table, but omits LALR(1) conflict resolution. Instead, it chooses one action (arbitrarily) to put in the parse table, and puts the other action in a second so-called 'conflict' table to be available to the parser driver at runtime.

When a Blender language description incorporates more than one lexical description, all of them are combined. In each description, any condition states declared (including the default initial state) are tagged with their language name to ensure their uniqueness. All rules are then merged into a single list of rules. Each rule whose condition state was not explicitly declared is now declared to belong to the tagged initial condition state for its language. The default lexical description's initial condition state is made the initial condition state of the combined specification. Rules that were declared to apply to all condition states (denoted by $<\star>$ at the beginning of the rule) are subsetted to apply only to those states declared for that particular language. This state-renaming scheme avoids any problems that the reordering of the rules may cause to the semantics of each language's lexical specification.

However, now each embedded lexical description's initial condition state is disconnected from the new initial state. It falls to the parser to set the lexer state before each token is lexed. For each parse state created by the GLR parser generator, the lexical descriptions to which the shift and reduce lookahead terminals belong are determined. This information is written into a table mapping a parse state to a set of lexical description IDs. At runtime, as the parser analyzes a document described by an embedded language description, it uses this table to switch the lexer instance into the proper lexical state(s) before identifying a lookahead token. If there is more than one lexical state for a particular parse state, the parser has to tell the lexer instance to switch into *all* of the indicated lexical states. However, any parse state that has more than one lexical state causes the input stream to become ambiguous. The analysis of this ambiguity is described in the next section.

5.4.7 Parsing Embedded Languages

Embedded languages add to the variety of input stream ambiguities described in Section 5.2 by enabling the lexer and parser to simultaneously analyze the input with a number of logical language descriptions. We can support embedded languages with one change to the XGLR algo-

[9] Since any context-free grammar can be parsed using GLR, merging causes no difficulty for the analyzer.

```
SETUP-LEXER-STATES()
for each pair of parse states p, q ∈ active-parsers list
  if lexer state of lex_p = lexer state of lex_q
    set lex_p to copy lex_q
```

```
for each parse state p ∈ active-parsers list
  let langs = lexer-langs[p]
  if |langs| > 1
    let each of q_1 .. q_n = copy parse state p
    for each parse state q_i ∈ q_1 .. q_n
      if langs_i ≠ lexer language of lex_p
        set lex state of lex_{q_i} to init-state[langs_i]
    add q_i to active-parsers list
  else if langs_0 ≠ lexer language of lex_p
    set lexer state of lex_p to init-state[langs_0]
```

Figure 5.11: An update to *SETUP-LEXER-STATES()* to support embedded languages.

rithm presented above.

In Figure 5.11, we see a modified version of *SETUP-LEXER-STATES()*. Before lexing the lookahead token for each parser in *SETUP-LOOKAHEADS()*, *SETUP-LEXER-STATES()* looks up the lexical language(s) associated with each of the parse states in the *active-parsers* list. If the language has changed, the state of the parser's lexer instance is reset to the initial lexical state of that language (via a lookup table generated by Blender). When there is more than one lexical language associated with the parse state, it implies that there is a lexical ambiguity on the boundary between the languages. This situation is handled in the same way as the other input stream ambiguities: for each ambiguity, a new parser is forked, and its lexer instance is set to the initial lexical state of that language. Each lexer instance will then read the same characters from the input stream but will interpret them differently because it is in a different lexical state.

The complete XGLR parsing algorithm which supports both ambiguous input streams and embedded languages can be found in Appendix C.

5.5 Implementation Status

Performance measurements of the parser are dependent on the nature of the grammar used and the input provided. In Spoken Java, punctuation is optional. Consequently any number of implicit punctuation symbols (*e.g.* comma, period, left paren, right paren, quote) must be considered between any two identifiers. This blows up the number of ambiguities during parsing to astronom-

ical levels for an entire program. In contrast, the possible lexical ambiguities in the specification rarely increase the ambiguity of the language dramatically, since they typically correspond to only a few structural ambiguities. In practice, the user interface limits the input between incremental analyses to 30 or 40 words. Limiting the input in this way makes the parse time tractable, even though it results in a large number (tens) of ambiguous parses. When filtered by semantic analysis, the number of semantically valid parses drops to a small number, usually one.

5.6 Related Work

Yacc [46], Bison [17, 22], and their derivatives, which are widely used, make the generation of C-, C++- and Java-based parsers for LALR(1) grammars relatively simple. These parsers are often paired with a lexical generator (Lex [58] for Yacc, Flex [75] for Bison, and others) to generate token data structures as input to the parser. Improvements on this fairly stable base include GLR parser generation [84, 95], found in ASF+SDF [50], and more recently in Elkhound [67], D Parser [76], and Bison 1.50. Incremental GLR parsing was first described and implemented by Wagner and Graham [104, 107, 108] and has been improved in the last few years by our Harmonia project.

There has been considerable work in the ASF+SDF research project [50] on the analysis of legacy languages, as well as language dialects. One central aspect of this work increases the power of the analyses by moving the lexer's work into the parser and simply parsing character by character. Originally described as scannerless parsing [88, 89], this idea has been adapted successfully by Visser to GLR parsing [99, 100]. Visser merges the lexical description into the grammar and eliminates the need for a special-purpose analysis for ambiguous lexemes. Some of the messiness of Flex interaction that we describe for embedded languages can be avoided. In making this change, however, some desirable attributes of a separate regular-expression-based lexer, such as longest match and order-based matching, are lost, requiring alternate, more complex, implementations based on disambiguation filters that are programmed into the grammar [98].

In the Harmonia project, a variant of the Flex lexer is used – historically because of the ability to re-use lexer specifications for existing languages, but more importantly, because a separate incremental lexer limits the effects an edit has on re-analysis. In Harmonia's interactive setting, the maintenance of a persistent parse tree and the application of user edits to preexisting tokens in the parse tree contribute heavily to its interactive performance. For example, a change to the spelling of an identifier may often result in no change to the lexical type of the token. Thus, the change can be

completely hidden by the lexer, preventing the parser from doing any work to reanalyze the token. In addition, the incremental lexer affords a uniform interface of tokens to the parser, even when the lexer's own input stream consists of a variety of characters, normal tokens and ambiguous tokens created by a variety of input modes.

In principle, both incrementality and the extensions described in this paper could be added to scannerless GLR parsers. However, as always, the devil is in the details. In an incremental setting, parse tree nodes have significant size because they contain data to maintain incremental state. If the number of nodes increases, even by a linear factor, performance can be affected. More significantly, incremental performance is based on the fact that the potentially changed region of the tree can be both determined and limited prior to parsing by the set of changed tokens reported from the lexer. For example, only a trivial amount of reparsing is needed if the spelling of an identifier changes, since the change does not cross a node boundary. Although we have not done a detailed analysis, our intuition is that without a lexer, the potentially changed regions that would end up being re-analyzed for each change would be considerably larger.

Aycock and Horspool [6] propose an ambiguity-representing data structure similar to our AmbigNode. They discuss lexing tokens with multiple lexical types, but do not discuss how to handle other lexical ambiguities. Their scheme also requires that all token streams be synced up at all times (inserting null tokens to pad out the varying token boundaries). Our mechanism is able to fluidly handle overlapping token boundaries in the alternate character streams without extraneous null tokens.

CodeProcessor [97] has been used to write language descriptions for lexically embedded languages. CodeProcessor also maintains persistent document boundaries between embedded documents. Gould et. al. [34] describe a static analysis of potentially dynamically generated SQL query strings embedded in Java programs. Specialized fragment analyses are likely to be required to semantically analyze this kind of embedded language.

Synytskyy, Cordy, and Dean [94] provide a cogent discussion of the difficulties that arise with embedded languages, and describe the use of island grammars to parse multi-language documents. They also summarize related research in the use of coarse parsing techniques for that purpose. Unlike the approach we have taken, they handle some of the boundary difficulties, such as those concerning whitespace and comments, by a lexical preprocessor prior to parsing.

5.7 Future Work

Blender, our lexer and parser generator, is built using language descriptions for its Flex and Bison variant input files. Flex, in particular, is made up of three languages: the Flex file format, regular expressions, and C. The three languages combine to form several kinds of interesting ambiguities. First, whitespace forms the boundary between regular expressions and C code in each Flex rule. In many parser frameworks, whitespace is either filtered by the lexer, or discarded by the parser, but certainly not included in the parse tables. However, in this case, whitespace must be considered by the parser in order to properly switch among lexical language descriptions at runtime. Second, whitespace takes on additional significance in Flex since rules are required to be terminated by carriage returns, even though carriage returns *are* allowed as general whitespace characters within rules. Third, it is possible to have non-obvious shift-shift conflicts between multiple interpretations of the same character sequence because they are interpreted in different lexical descriptions. For example, the following is the actual grammar production for Flex rules (first described in Section 5.4.5):

RULE $\longrightarrow$ STATE? REGEXP_ROOT$_{regexp}$ WSPC CCODE

STATE $\longrightarrow$ < ID >

The optional STATE can begin with a $<$ token. But $<_{regexp}$ is a valid regular expression token as well. Since the STATE is optional, the $<$ character may be lexed as two separate tokens, leading to a lexical ambiguity. However, the Flex manual states that if a Flex rule begins with a $<$, it must be the beginning of an optional STATE, not a regular expression. If the input is not actually a proper state, it is an error, not a regular expression. We are currently upgrading our language analysis technology and the grammar transforms used in Blender to handle these three kinds of ambiguities.

New techniques being developed in our research group for batch GLR parser error recovery do not yet take into account the ambiguities discussed in this paper. Extension of the work above to incorporate batch error recovery is ongoing. Incremental error recovery is change-based and has already been extended.

Automated semantic disambiguation of both homophones and syntactic ambiguities requires integration with name resolution and type checking. In addition, to handle ambiguities that arise in an interactive setting (*e.g.* via edits in a program editor) semantic information must be persistent and incrementally updateable. Such persistence will enable analysis of edits to a portion of the program to use semantic information from surrounding code to help disambiguation (for example, by providing a list of all legal visible bindings at the edit location). A MultiText identifier

token appearing in a variable use position can be disambiguated if one of its alternatives matches a definition that is in scope and has the right static type. Our solutions to these problems are presented in Chapter 6.

Chapter 6

The Inheritance Graph – A Data Structure for Names, Scopes and Bindings

Automated approaches to generating program analyses have a long history. Lexer and parser generators are well-known and commonly used in production compilers and program analyses. Machine code is also commonly generated. Name resolution, type checking and optimization, however, are not often automated.

In this chapter, we present a formalism for name resolution, called the Inheritance Graph (IG) (originally described in the Ph.D. dissertation of Phillip Garrison [33]), that we have extended and implemented to address disambiguation of spoken input in a program. The IG is powerful enough to describe all programming languages' name visibility rules. It is a graph-based data structure which represents the names, scopes and bindings found in a program. The nodes of the graph correspond to scopes in the program. At each node, each name defined in that scope is stored in a binding with its declared kind and type. A directed edge between two nodes means that any names that are bound in the origin node are also visible from the destination node, *i.e.* the name defined in the origin node is "inherited" by the destination node. Inheritance is transitive – bindings flow from node to node along the edges. To localize visibility, bindings that are inherited can be copied into the destination node. This process is called propagation. If two bindings with the same name both reach a node, they are said to "clash." Languages vary in their clash resolution rules.

When propagation has finished, each node in the graph has a list of bindings which are

visible from that point in the program. The design of the IG for a given language may prevent all visible bindings from flowing to that node. If all visible bindings reach a node, then name lookup can occur in constant time, despite the complexity of the name or the lookup rules.

The information stored in the graph's nodes is persistent, and is incrementally updated as the program changes. A syntactic differencing algorithm is run over the parse tree after an edit, revealing the places in the program where names may have been added, deleted or changed. Once the altered names have been used to update the IG nodes and bindings, the graph is re-propagated to populate all nodes with the latest set of visible bindings.

The IG can be used in place of a symbol table or attribute grammar for type checking. We use it for spoken language disambiguation in a programming-by-voice system. The ability to list all names visible at a particular program point can be used to restrict the vocabulary available to the voice recognizer, improving recognition accuracy. Fast name lookups also enable quick verification of the many ambiguous interpretations of program input found when speaking programs. The IG's persistence enables fast access to semantic information and documentation which many popular IDEs, such as Eclipse, present to programmers via text hovers in the editor. Listing all names visible at a particular program point facilitates implementation of code completion tools.

In the rest of this chapter, we describe the IG data structure in greater detail. First, we summarize concepts in name resolution and visibility, all of which can be modeled using the IG. Section 6.2 presents a small Java example and its IG. That example is used in Section 6.3.1 to show how the graph can be designed and constructed. Section 6.3.2 explains the binding propagation algorithm and clash function which spreads names and bindings throughout the graph. The next section explains incremental update of the IG after each program edit. Afterwards, we show how the graph can be applied to type checking, programming by voice and interactions in IDEs. In Section 6.7, we discuss the implementation of the IG. Finally, we present related work.

6.1 Survey of Name Resolution Rules

Programming languages exhibit a large variety of name visibility rules, ranging from fairly simple (*e.g.* Fortran) to downright complex (*e.g.* Java). We begin this section with a number of definitions of the terminology used, and then survey name resolution and visibility control rules for a wide variety of programming languages.[1] The concepts shown here will be used in the next

[1] This section is a summarization of Chapter 2: Survey of Visibility Control in Programming Languages which appears in Phillip Garrison's dissertation [33].

few sections in our discussion of the IG and its realization for the Java programming language.

6.1.1 Definitions

A *name* is a literal string of characters used to denote an entity in a program. This name is usually found in programs as identifiers or labels, for example the name of a variable or function. A *qualified name* is a sequence of names strung together to form a more specific reference to a single name found in the program.

An *entity* is a type, constant, or other program object that is involved in the semantics of the program. In Java, there are several *kinds* of entities: classes, interfaces, fields, methods, constructors, local variables, packages, labels, and catch clauses.

A *binding* is a mapping between a name and an entity. The name resolution phase of program analysis is concerned with computing the bindings between all names and entities found within a program. It is often possible to determine the kind of a name reference directly from its syntactic position in the program.

A *scope* is a region of a program where a binding is visible. Early languages such as Fortran had only two notions of scope, local to a function, or global to the program. Algol 60 introduced block structure which defined nested scopes of visibility. A name defined in an outer block can be *inherited* and made visible to an inner block, but not vice versa. In many languages, scopes can be named, for example a class declaration or record type.

When two bindings that are visible in a scope have the same name, they are said to *clash* in that scope. For example, defining two variables with the same name in a block in Pascal is illegal. In most languages with nested blocks, a variable binding defined in an inner block with the same name as in an outer block *shadows* the binding of the outer variable name.

Each programming language has a specific set of *visibility rules* which define how names are declared and visible throughout a program. Some languages such as Fortran have very simple visibility rules, while others (*e.g.* Java) are much more complex. The rules affect all aspects of name visibility: how names are declared and implicitly defined in various scopes, how to resolve names, and how names are declared and defined in explicitly named scopes.

6.1.2 Implicit Name Declarations

Names are usually introduced through explicit declarations, such as a function or variable declaration. Since almost all languages support nested scopes, it is important to define to which

scope a declaration in a series of nested scopes belongs. Many languages associate declarations with the innermost scope in the series of scopes in which they appear. PL/I, however, assigns implicit name declarations to the global scope.

When a binding is added to a scope, it may clash with another binding with the same name. Clashes can be resolved in one of five ways: (1) No actual clash: Identical names may be associated with different namespaces. For instance, functions and variables might have independent namespaces and can never clash. (2) Shadowing: The new binding overrides the old binding and takes its place. This is common in languages with block structure. (3) Overloading: A single name is bound to more than one entity, requiring further semantic information to distinguish between them. Some languages support overloading of functions based on their argument types. (4) Reference: A new binding is an actual definition of a prior declaration. (5) Error: Two bindings may not have the same name.

Languages with single-pass compilers such as C and Pascal often have a declaration-before-use rule, which means that the declaration of a name must appear textually before its use. In C, programmers declare prototype functions and variables in header files in order to conform to this rule. Java requires declaration before use only for fields, and for variables declared inside method bodies. Lisp employs more complex declaration-before-use rules with its `let`, `let*` and `letrec` constructs. Each of the three forms allows the introduction of more than one variable, but differ in the use of those variables in the variables' initializers. `let` is the most restrictive – no variable defined in the construct may be used in any variable's initializer. `let*` defines that each variable initializer executes in the order that it appears, allowing variables that textually precede the other to be used in the other's initializer. `letrec` relaxes all rules and allows all variables to be used in each other's initializers.

Names can be created and bound to their entities at compile-time, load-time, or run-time. Some languages such as Lisp, Logo or Snobol allow programs to create and bind names at runtime. Languages that support run-time binding such as Lisp or Smalltalk do so to enable a flexible coding style, but often encounter criticism from proponents of statically-typed, statically-bound languages where type errors can be caught before applications ship to customers. Traditionally, languages with dynamic name creation have been implemented with interpreters. Dynamic binding is typically associated with languages that also support dynamic name creation.

6.1.3 Name Resolution

Once names have been declared and bound, uses of those names must be resolved to the proper binding. A name lookup may find a binding in the local scope, or may require searching several scopes. Each programming language defines a set of search rules for discovering names within the scopes of the program.

Block structured languages usually rely on lexical (or textual) scoping rules for name lookups. In this form, the lookup of a name is closely tied to the syntactic structure of the program. If one block is nested within another, the outer block's names are visible within. Dynamic scope uses the notion of runtime nesting to determine the search order for a name. For the most part, this mimics lexical scope, except when a function has a free variable, one that is not declared within the function. In dynamic scoping, a definition of that variable name in a scope that encloses the function *call* site propagates into the function definition. In a lexically-scoped language with closures (pointers from functions to the scopes in which they were created), the closure pointer for a function is searched instead, enabling definitions of names that lexically enclose the function *definition* site to propagate into the function.

Some languages have a type-safe way to delay name to type binding until runtime. In Algol 68, one could statically bind a name to a union of types, and use a type coercion function at runtime to downcast it to one of the union members in a type-safe way. This is similar to C's use of untagged unions, but without the type unsafety. Object-oriented languages formalize a variant of this mechanism through their inheritance hierarchy. A name can be bound to a superclass and downcasted to the particular subclass right before its use.

6.1.4 Explicit Visibility

Most languages provide named scopes to control visibility of the names declared within (*e.g.* classes in an object-oriented language, named records in Pascal). To enable access to named scopes, qualified references are provided. A qualified reference is a sequence of name references that is looked up in order. The binding of the first name points to another named scope which is used to look up the second name, and so on. Qualified references enable a hierarchical organization for names in the program, but still enable simple lookups for local names.

A module is a collection of types, functions, variables, records, or other modules used to define a more general kind of named scope. Modules are known by various names in programming languages: namespaces in C++, packages in Ada and Java, modules in Modula 2 and Modula 3. In

object-oriented languages, modules can be extended into objects by allowing them to have multiple instantiations. In C++, classes can only be defined statically, but in Smalltalk they can be created on-the-fly and treated as first-class objects themselves.

Every language has a mechanism for declaring access control to names. For example, in C++ all names defined in structs are by default publicly exported to all outside users of the struct, but in classes, all names are by default private. Java has four kinds of access: public (available to all users, private (available only inside members of this class), protected (available to this class and its subclasses), and the default, package protected (available to all classes within the same package). Another area of customization is whether access control is absolute or may be overridden by another declaration.

In object-oriented languages, classes can be associated with one another via inheritance. In single-inheritance languages, like Smalltalk and Objective-C, a class may only have one superclass, from which it inherits the names of public and protected methods and fields. In languages with multiple inheritance, such as Common Lisp and C++, classes may inherit names from any number of superclasses. This makes name lookups more complex. In Lisp, all superclasses of a class are sorted topologically; name lookup proceeds in topological order through the superclasses, successful even when the inheritance hierarchy paths have a common ancestor. C++ has no such precise ordering, which can cause many problems at runtime, including duplicate execution of methods when superclass hierarchies have common ancestors.

Name-entity bindings in *open* scopes are visible to other scopes which are lexically or dynamically nearby. Some languages like Euclid and Modula support *closed* scopes, into which outside names can not propagate. Names can be imported into a scope via an import statement. Import statements reference a name or module from another scope and makes that name or module's names available as local names. In Java and C#, closed scopes are used to simplify the qualified names required in a program that uses many different classes defined in many different packages. Export statements are the inverse of imports; they make names from one scope available to an enclosing scope.

The import of a whole module may cause too many names to be imported into a scope, causing name clashes. A language that allows imports must have a rule for resolving name clashes between imported and local names. In Ada, if two imported names clash, neither are visible, unless they are overloadable, in which case only one needs to be visible within the scope anyway. In Java, if two imported names clash, uses must refer to both entities via fully qualified references, rather than using their single name.

```
int fib(int n) {
    if (n < 2) {
        return 1;
    } else {
        int fib_n_minus_1 = fib(n - 1);
        int fib_n_minus_2 = fib(n - 2);
        return fib_n_minus_1 + fib_n_minus_2;
    }
}
```

Figure 6.1: A small Java method with multiple local scopes.

In a few languages, an aliased entity may be known by more than one name. This can be confusing to users, who expect that each name will refer to a unique entity. Aliasing is possible in Fortran and PL/I through overlay mapping of names and types to areas of storage in memory. In Euclid, one can rebind an entity to a new name. Within the scope of the bind statement, the old binding may not be referenced in favor of the new one. Ada and Algol 68 have similar rebinding constructs. Rebinding is also used in dynamic compilation environments; a value may be tagged as a constant expression whose value is recomputed every time a function is entered (causing the function to be recompiled and optimized based on the update value).

6.2 Java Inheritance Graph Example

Next, we describe the Inheritance Graph (IG) we have designed to support name resolution for the Java programming language. Throughout the exposition, we will refer to the fragment of Java code shown in Figure 6.1.

An IG node represents a scope in the source language. Each binding in a node has a label designating a Visibility Class. Visibility classes (VCs) also label most edges. The VCs control the flow of bindings from one node to the next. When a binding in a node has the same VC as an outgoing edge from that node, it may flow across that edge into the next node.

In the example, the *fib()* function contains three obvious scopes, one for the method body, and two for the branches of the `if` statement. There are actually two more: Java introduces new scopes for each local variable declaration (so as to properly implement def-before-use). This method body is represented by five IG nodes connected by the directed edges shown in Figure 6.2. There are two nodes for the method declaration.

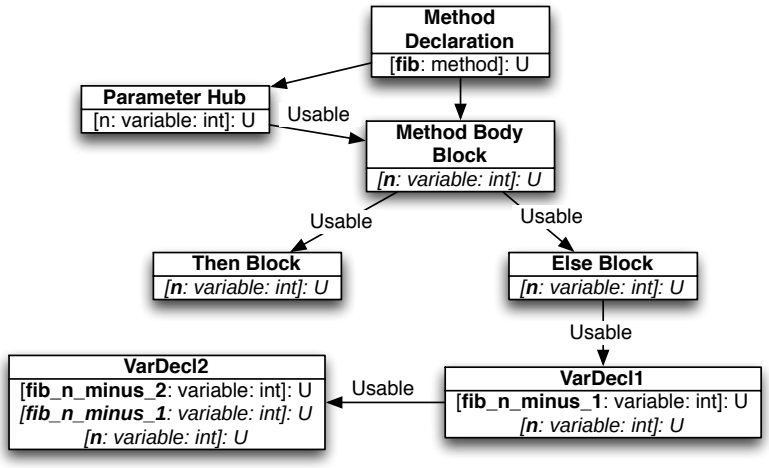


Figure 6.2: The IG subgraph for the Fibonacci method. A binding is marked in brackets and is a tuple of name, kind and type. The letter after the binding is the binding's Visibility Class, also shown in long form labeling the edges. Local bindings are shown in a normal font. Inherited bindings are in italics.

All names defined in a scope are added as a name-entity binding stored in a set of local bindings in that scope's IG node. In *fib()* there are two local variables defined in their respective *VarDecl* IG nodes. A Java entity is a pair of a kind and a type. Both `fib_n_minus_1` and `fib_n_minus_2` are *variable* kind of type `int`.

The last line of the method uses both variables, and is located in the same scope as the second *VarDecl*. In order to resolve both variable names properly, `fib_n_minus_1` must be visible in the second *VarDecl* node. All variable bindings are labeled with a VC called *Usable*. The edges between blocks' IG nodes representing a Java method are also labeled with that VC. Thus, the binding for `fib_n_minus_1` flows from the first *VarDecl* node into the second. During type checking, when the variable names are looked up for their use in the plus expression, they are found in the node containing the plus expression. Their types, both `int`, are verified to be compatible with + and the expression type checks properly.

The method body is attached to the method declaration IG node, which contains a binding that links the name of the method *fib()* to its kind *method* and type signature *int* → *int*. Attached to the method declaration node is a parameter hub node where all of the method parameters are bound. Method parameters have the same kind as local variables, and employ the same VC, *Usable*. The method parameter hub is connected directly to the method body with an edge labeled *Usable* so that the method parameters can flow into the method body. Hub nodes are a design pattern used in the

Inheritance Graph to designate a high-degree node. It is possible to find any IG node in the graph by knowing to which hub node it should be attached.

Note that the edge from the method declaration node to the method body is not tagged with *Usable* even though the method name is. Resolving a method name is not as easy as having it flow into the node where it can be used, due to method overloading and the interaction between the lexical scoping of classes and object-oriented inheritance. This issue is discussed in Section 6.6.1.

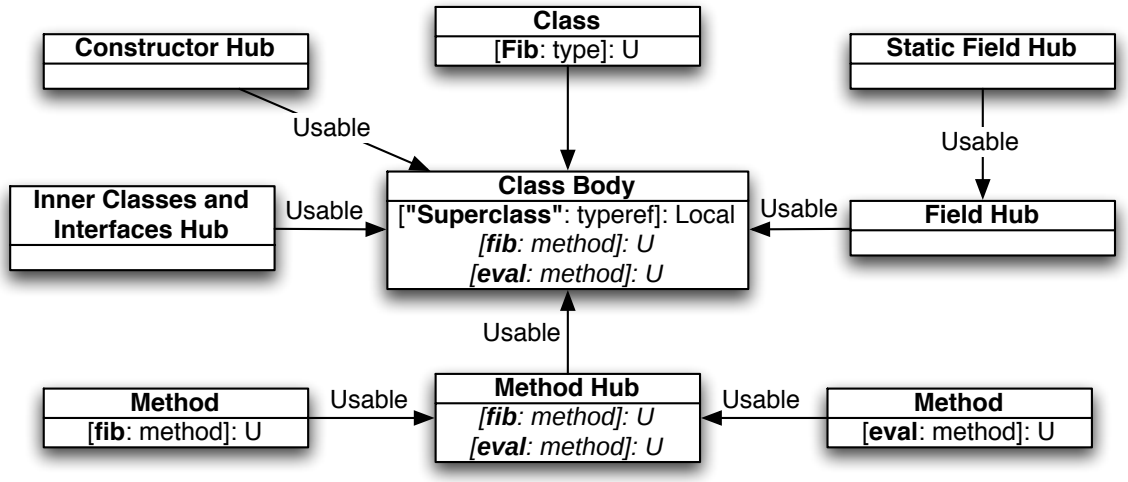


Figure 6.3: The IG for the Fib class.

A class in Java contains fields and static fields, methods and constructors, and nested classes and interfaces. They are represented by several IG nodes (shown in Figure 6.3): one each for the class declaration and class body, and one node each for a hub for fields, static fields, methods, constructors, and nested classes and interfaces. The method *fib()* that was defined earlier is attached to the method hub node. Other methods in the class (in our example, *eval()*) are also attached to the method hub. Fields and static fields are attached to their respective hubs in a linked list in order to preserve their ordering in the file (declaration order indicates initialization order at runtime). Constructors and nested classes and interfaces are also connected to their own hubs. All of the methods, constructors, fields, static fields and nested classes and interfaces are bound with a *Usable* VC which enables them to flow into the class body.

Compilation units contain classes via a Class and Interface Hub node. Packages contain compilation units via a package body IG node. All class bindings in a package flow via *Usable* into the package body node. Packages are connected in a nested fashion (even though their semantics do

not work that way) to the root node of the Java IG.

There are several other VCs used in the Java IG which pass lexical information downward through the graph from the packages to the method bodies. Examples include *Reference* and *Local*. *Reference* bindings represent the closest enclosing class or interface, the compilation unit to which each class belongs, and the package to which each compilation unit belongs. At every IG node, these reference bindings are available to help with name lookup (rather than propagating all visible bindings directly to the node). The superclass and superinterfaces for a class or interface are stored in the class or interface body node with VC *Local*, and do not propagate anywhere else. The class itself is the closest enclosing class for its contents, so by resolving the closest enclosing class, it is possible to find the superclass and superinterfaces with one extra indirection.

6.2.1 Propagation

Each of the VCs enables a certain set of bindings to flow around the graph. When these bindings have the same name, they clash. As explained in Section 6.1.1, one of five resolutions to the clash is possible. In Java, if the two names are a method and a field, then there is no clash because these names do not exist in the same namespace. If they are two of the same kind, they either overload, shadow or there is an error. Method and constructor names overload (unless they have identical type signatures, in which case it is an error). Package, class, interface and field names may not be duplicated – this is always an error. When two references (such as two enclosing scopes) clash, the inner binding shadows the outer binding.

6.2.2 Name Lookup

Name lookup in Java is very complex. Names can be single or qualified, and composed of package, class, interface, field and method single names. A given entity may be referenced by many names depending on where the reference occurs. In fact, the kind of the reference also affects how it can be resolved, which presents difficulties to the IG model that were not anticipated when it was designed. Because of these complications, lookups are not done simply by finding the appropriate binding in the local node. Java lookups are discussed in Section 6.6.1.

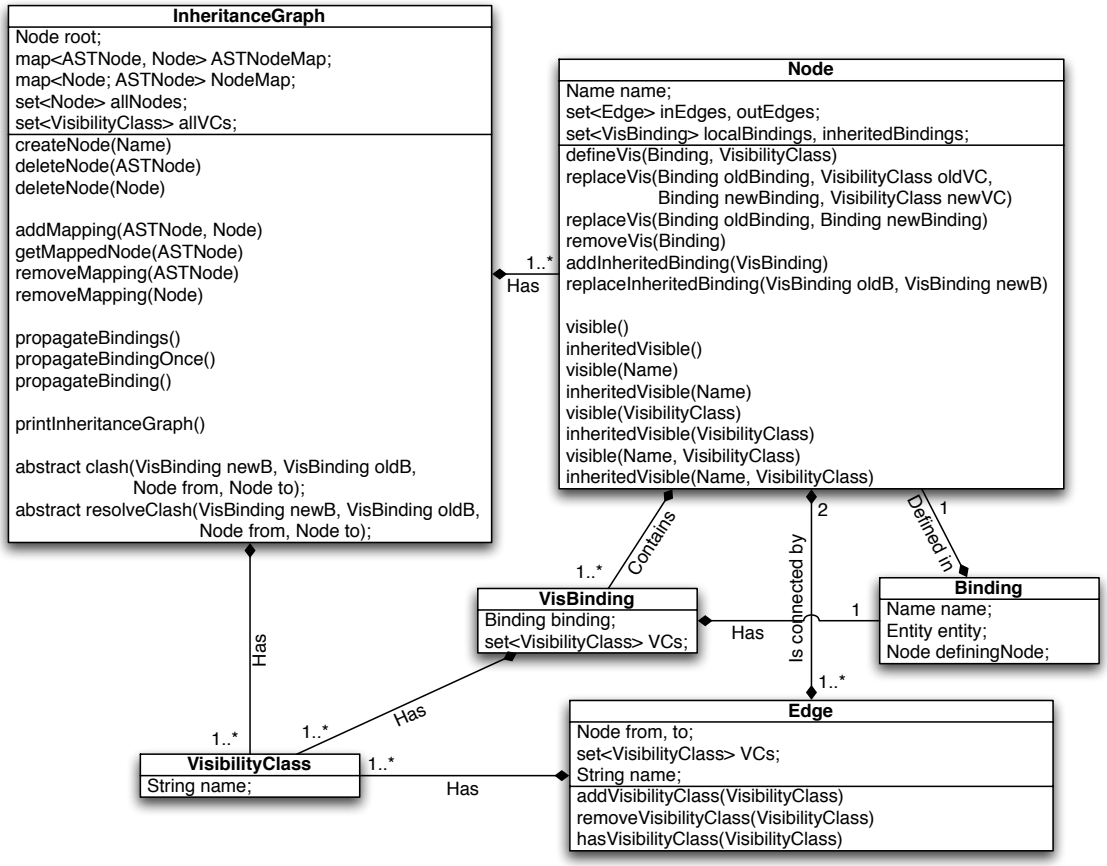


Figure 6.4: UML diagram for the IG. A Graph is made up of a connected set of nodes and directed edges. Each node contains sets of VisBindings, which define the names in the program that are defined or visible there. A VisBinding is a pair of a Name-Entity Binding and a set of Visibility Class labels. These Visibility Class labels are also used to label edges in the graph. VisBindings flow along the directed edges when at least one of their visibility classes match the ones on the edge.

6.3 The Inheritance Graph Data Structure

The IG (see Figure 6.4) is composed of a language-independent graph data structure combined with a language-dependent extension. The extension defines the types for names, entities, AST nodes, and the particular VC labels used for a given language.

A node contains a set of in-edges and out-edges that point to other nodes. An edge is a directed connection between nodes that is labeled with a possibly empty set of VC labels. Edges may be labeled by more than one VC. There are also two collections of binding information, one

for bindings declared at that node (local bindings) and the other for bindings inherited by the node during binding propagation (inherited bindings). A binding is a triplet of a name and entity and the node where that binding was locally defined. When added to a node, a binding is further paired with a set of visibility classes drawn from the same set used for edges. Each node and edge has a unique name to easily identify it during any manipulation of the graph structure.

In addition to the collection of nodes and VCs in the IG, there is a bi-directional map from AST node to IG node. This map enables parse tree visitors to easily jump to the portion of the IG corresponding to a visited AST node. The reverse mapping is useful for updating IG nodes when the user's program has been edited.

6.3.1 Graph Construction

The IG for most languages can be constructed purely syntactically. Each language has a finite set of program constructs that represent scopes in the program. For example, in Cool, a programming language used in an undergraduate compiler class at UC Berkeley, there are nine kinds of scopes: the entire program, each class, each method and field, each let and let binding, each type case and case binding, and each anonymous inner scope inside a method body. It is easy to write down a tree walk of the parse tree that will create IG nodes for each construct.

As each IG node is created, it must be anchored into the growing graph. The graph has a root node (for Cool, it will be the node corresponding to the program itself), to which top-level program entities (packages, namespaces, classes) will be attached. For each named entity in the program, a binding is created and stored in the appropriate IG node's list of local bindings. When several names are defined (*e.g.* method parameter lists, fields and methods in a class) in a single scope, each name gets its own binding in the node's local bindings. Every binding is associated with a set of VCs to determine how it will propagate through the graph.

6.3.2 Binding Propagation

Once all of the bindings from the program are stored in the IG, they must be propagated to all reachable nodes compatible with their VCs until a fixed point is reached.

In the propagation algorithm (shown in Figures 6.5 and 6.6), the inherited bindings set for each node is initialized to the value of the local bindings set. This forms the start of propagation. Next a work list of nodes to process is created. As each node is processed, it is removed from the work list. If a node changes due to propagation, it is re-added to the list. The work list is

```
PROPAGATE-BINDINGS(NodeList nodes)
foreach node ∈ nodes
  if not already copied(node)
    set inherited bindings of node to local bindings of node
init work list to nodes
while work list ≠ ∅
  set work list to PROPAGATE-BINDINGS-ONCE(nodes)

PROPAGATE-BINDINGS-ONCE(NodeList nodes)
init work list to empty
foreach node ∈ nodes
  if node ∈ work list
    remove node from work list
  foreach out edge ∈ out-edges of node
    foreach visible binding ∈ inherited bindings of node
      let VCs = (visibility classes of visible binding) ∩ (visibility classes of out edge)
      if VCs≠ ∅
        let to node = to node of out edge
        let new VB = (binding of visible binding) × VCs
        if PROPAGATE-BINDING(node, new VB, node)
          // a binding propagated into the to node
          add to node to work list
return work list
```

Figure 6.5: Binding propagation algorithm Part 1.

initially unsorted; a sorting function on the nodes may improve the constant factor of the fixed point computation.

In *PROPAGATE-BINDINGS-ONCE()*, for each node, the algorithm processes all inherited binding $\times$ out-edge pairs to see which bindings have matching edge labels. If there are any labels in common, the inherited binding (actually a copy which only includes the matching edge labels) is propagated to the target of the out edge. Once at the target node, the algorithm checks (in *PROPAGATE-BINDING*) whether the binding was already propagated there. If not, it tests for a clash.

Two bindings clash in a language-specific way, but usually because their names are identical and their kinds are compatible (*i.e.* the names are in the same namespace). For example, in Java, class names and method names may be identical, but can never clash because syntactically they ap-

```
PROPAGATE-BINDING(Node from node, Node to node, Visible Binding new visible binding)
foreach old visible binding ∈ inherited bindings of to node
  if (binding of old visible binding) ≠ (binding of new visible binding)
    if CLASH(old visible binding, new visible binding, from node, to node)
      let winning binding = RESOLVE-CLASH(old visible binding, new visible binding,
                                          from node, to node)
      replace old visible binding ∈ inherited bindings of to node with winning binding
  else break // new visible binding cannot clash because it is already present
let added? = add new visible binding to inherited bindings of to node
return added?

CLASH(Visible Binding old visible binding, Visible Binding new visible binding,
      Node from node, Node to node)
// Programming Language Specific Code
// Returns a boolean if these two bindings actually clash

RESOLVE-CLASH(Visible Binding old visible binding, Visible Binding new visible binding,
              Node from node, Node to node)
// Programming Language Specific Code
// Returns either the old or new visible binding
```

Figure 6.6: Binding propagation algorithm Part 2.

pear in different contexts. The clash logic for a programming language is contained in the *CLASH()* function defined by the language designer. When two bindings are determined to clash, only one may win (for example, in many languages a local scope binding beats an outer scope binding). *RESOLVE-CLASH()* is another language-specific function that returns the winning binding.

During binding propagation, *RESOLVE-CLASH()* may detect errors in the program, such as two entities with the same name declared in the same scope. Errors must be propagated back to the program to be reported to the user. In our uses of the Inheritance Graph, we have stored pointers to AST nodes in the binding's entity. When a binding proves erroneous, its entity's AST node is used to identify the program location where the error message is put.

Two important invariants must be respected with the design of the IG visibility classes and bindings. First, binding clashes are transitive. If binding A beats binding B and binding B beats binding C, then binding A *must* beat binding C. Second, if two bindings clash, the winning binding must propagate to the same set of nodes as each binding would have in the absence of a clash.

Actually, we can be more relaxed about this – the set of nodes to which the losing binding would propagate in the absence of the winning binding must be a *subset* of the nodes to which the winning binding would propagate. This is required to ensure that propagation of bindings is idempotent and independent of the order of processing of the nodes.

Binding propagation has a time complexity of $O(bn^3)$, the number of bindings times the cube of the number of nodes in the graph. Each node stores two sets of bindings. The size of the first set is proportional to the number of bindings locally declared at that IG node. The size of the second set is proportional to the number of bindings in the entire graph. While true in principle, in practice the sets are much smaller, except in languages without hierarchical structure. Most languages limit the visibility of names to a local region where that name was declared. Most entities are accessed by hierarchical names, which enable partitioning of names into scope regions, with few apportioned to the global scope.

6.4 Incremental Update

The IG is a component of Harmonia, which employs an incremental lexer and incremental parsing framework to maintain a persistent parse tree representation of the program code being edited. The parse tree is robust to errors and incompleteness in the user program. All edits to a user's program are translated into edits on the parse tree, which then undergoes reanalysis to incorporate the changes. All edits and tree modifications are stored in a set of database-like data structures that maintain a complete edit history of attributes for each node in the tree [105].

When an edit results in a change to the parse tree, the Inheritance Graph must be brought up-to-date. The framework includes an algorithm which, when given two versions of a parse tree, can produce a set of syntactic differences that can be used to update the IG. This algorithm is generated from the grammar of the language once it has been annotated with tags that indicate how each production affects the IG.

6.4.1 Update Tags

The production below describes a class definition in Cool. Cool classes have a name `TYPEID`, a single optional superclass `TYPE`, and a set of semicolon-delimited features, which are methods and fields. The rule has been annotated with tags that show how changes in the elements of the right hand side can affect the inheritance graph.

CLASS → CLASS *name*:TYPEID *super*:(INHERITS TYPE)? { *features*:(FEATURE ;)*} ;
⇒ *createsIGNode*, *affectsThisIG*{*name super features*} *ignoredByParentIG*

The tag *createsIGNode* is used on any rule that is directly associated with an Inheritance Graph node. For Cool, this tag is set on the program root, all class declarations, method and field declarations, anonymous blocks, let bindings and case bindings. These are the only nodes in which a change to the symbols on the RHS can cause a change in the inheritance graph node associated with that program element.

The elements of the rule that affect the inheritance graph are listed after the *affectsIGNode* tag. In the class declaration above, the name of the class, its list of superclasses, and the features (methods and fields) defined inside are marked as being able to cause changes to the IG. The name and superclasses are the attributes of a class that are stored in the class' IG node; if these are changed, the class IG node must be regenerated. The methods and fields are different, however. Changes in a field or method do not change the class IG node itself, but only change its in and out edges, thus they do not require the class IG node to be regenerated. To distinguish between cases where a change requires the regeneration of the production's IG node and one where it does not, the *ignoredByParentIG* tag is employed. Each RHS nonterminal that is labeled with *affectsIGNode* but that does not require its production's IG node to be regenerated if it has changed has its own RHS productions marked with *ignoredByParentIG*. Neither method nor field productions are shown above. Intuitively, one can think of changes in RHS elements marked with *ignoredByParentIG* as having no effect because the names defined inside do not escape the scope of their declaration. Method and field names are accessible only inside the class itself. Fields are private, and methods can only be accessed as part of a qualified name lookup.

6.4.2 Syntactic Difference Computation

Computing syntactic differences requires two tree walks shown in Figures 6.7, 6.8, and 6.9. The first discovers deleted portions of the tree, and the second discovers added and changed portions. The tree walk begins at the program root.

Changes are accumulated into three sets, *deleted-nodes*, *added-nodes* and *changed-nodes* via two tree walks. The first tree walk checks nodes tagged with *createsIGNode* to see if they themselves were deleted. If this node was not deleted, but it contains the *affectsIGNode* tag, its children are checked for changes. Harmonia uses an EBNF grammar for its rules, thus children may be simple symbols, star or plus sequences (zero or more, or one or more elements) or optional

```
COMPUTE-TREE-DIFF(Node root, GVID old-version, GVID new-version)
init deleted-nodes to empty
init added-nodes to empty
init changed-nodes to empty
COMPUTE-DELETED-NODES(root, old-version, new-version)
COMPUTE-ADDED-CHANGED-NODES(root, old-version, new-version)
// In a language-dependent way, process the changes
// to update the Inheritance Graph
PROCESS-DIFF(deleted-nodes, added-nodes, changed-nodes)
```

Figure 6.7: Syntactic Difference Algorithm Part 1. This part computes a syntactic difference of nodes in which changes might affect the Inheritance Graph.

symbols. Each type of child requires a custom check for deletion. For example, when a child representing a star or plus sequence has been deleted, it means the entire sequence was deleted. If it was not deleted, we check if any of the elements in the sequence were deleted. It is hard to maintain differences in the order and content of a sequence, so if any elements are deleted, the entire sequence is marked deleted as well. Similar checks are performed on optional children (if the child existed in the old version, but not the new version, it was deleted. If it did not exist in the old version, we do not need to check it for deletion) and ordinary children. If any of the children were deleted, the node is added to the *changed-nodes* set. If the parent is not tagged by *ignoredByParentIG* and it was deleted, we return true for the return value of this recursive check. Otherwise, we return false. A similar algorithm suffices for computing the nodes added between versions.

The syntactic difference computation runs in time linearly proportional to the number of edits made to the parse tree (not the size of the entire parse tree). Our incremental lexer and parser algorithms also run in time linear to the number of edits [8, 104, 108] which enables quick computation up to this point.

6.4.3 Semantic Difference Computation

Once all the syntactic differences have been computed, they have to be translated into semantic differences, differences that affect the IG. One property of our use of the IG is that every call to create an IG node and subsequent subtree is either independent of other calls or fully dependent on another call. Thus, if there is a call to create an IG subgraph for a Java class, all other calls to create nodes will be contained wholly within that subgraph or create a completely different subgraph.

```
COMPUTE-DELETED-NODES(Node node, GVID old-version, GVID new-version)
if node is tagged by createsIGNode
  if node does not exist in new-version
    add node to deleted-nodes list
    return true
if node is tagged by affectsIGNode
  if node has nested changes between old-version and new-version
    init any-nodes-deleted? to false
    foreach symbol ∈ affectsIGNode
      init child-deleted? to false
      let child = child node named symbol in node in version old-version
      if symbol is a + or * sequence?
        if child does not exist in new-version
          set child-deleted? to true
        else foreach element ∈ child sequence
          set child-deleted? to child-deleted? or
              COMPUTE-DELETED-NODES(element, old-version, new-version)
      else if symbol is optional?
        if child is in the tree in version old-version
          if child is not in the tree in version new-version
            set child-deleted? to true
          else set child-deleted? to
              COMPUTE-DELETED-NODES(child, old-version, new-version)
      else if symbol is ordinary?
        set child-deleted? to COMPUTE-DELETED-NODES(child, old-version, new-version)
      if child-deleted?
        add node to changed-nodes list
      set any-nodes-deleted? to any-nodes-deleted? or child-deleted?
    if node is not tagged by ignoredByParentIG
      return any-nodes-deleted?
    else return false
```

Figure 6.8: Syntactic Difference Algorithm Part 2. This part computes a syntactic difference of nodes in which deletions might affect the Inheritance Graph.

```
COMPUTE-ADDED-CHANGED-NODES(Node node, GVID old-version, GVID new-version)
if node is tagged by createsIGNode
  if node does not exist in old-version
    add node to added-nodes list
    return true
if node is tagged by affectsIGNode
  if node has nested changes between old-version and new-version
    init any-nodes-deleted? to false
    foreach symbol ∈ affectsIGNode
      init child-added? to false
      let child = child node named symbol in node in version new-version
      if symbol is a + or * sequence?
        if child does not exist in new-version
          set child-added? to true
        else foreach element ∈ child sequence
          set child-added? to child-added? or
              COMPUTE-ADDED-CHANGED-NODES(element, old-version, new-version)
      else if symbol is optional?
        if child is in the tree in version new-version
          if child is not in the tree in version new-version
            set child-added? to true
          else set child-added? to
              COMPUTE-ADDED-CHANGED-NODES(child, old-version, new-version)
      else if symbol is ordinary?
        set child-added? to COMPUTE-ADDED-CHANGED-NODES(child, old-version, new-version)
      if child-added?
        add node to changed-nodes list
      set any-nodes-added? to any-nodes-added? or child-added?
    if node is not tagged by ignoredByParentIG
      return any-nodes-added?
    else return false
```

Figure 6.9: Syntactic Difference Algorithm Part 2. This part computes a syntactic difference of nodes in which changes might affect the Inheritance Graph.

This suggests a simple update strategy. When a parse tree node is deleted or changed, delete the corresponding IG node and its subgraph and recreate it from scratch. Since in many languages, the only deleted, changed, or added nodes with any effects on the subgraph will be nested within that topmost node that has changed, one need not check any node inside a deleted subtree for deletions in the Inheritance Graph. Added nodes can be dealt with simply by adding the new IG nodes into the tree as during batch Inheritance Graph creation. If an AST node has been changed in a way that affects the created IG node itself, and none of its IG children, it can have its bindings brought up to date with the new information in the parse tree.

6.4.4 Repropagation

Once the IG nodes and local bindings have been brought up to date, the graph must be repropagated. Normally this requires rerunning the entire propagation algorithm on the entire Inheritance Graph, but several optimizations are possible. If a change causes the name of a binding to be altered, the graph can be updated simply by running over all nodes once to delete any inherited bindings with that name ($O(n)$), and then running propagation, but only on the old and new names and only on the nodes where the first name had propagated ($O(m^3)$). If the entity in a binding has been updated, another linear search of the graph is performed to delete any bindings with this entity, and then a similar repropagation is performed, but only on the name that was updated. Again, this is $O(m^3)$. If nodes must be added to the graph, consider A to be the set of nodes in the frontier of the graph that connects to B, the newly added subgraph. For names that propagate from A to B (from the old graph to the new subgraph), all of A's inherited bindings are propagated to B, and then propagation is run only in B (thereby reducing the number of nodes in propagation to the size of the change). For names in B that must propagate to A, all names in the old graph with names declared locally in B are deleted, and propagation from B to the old graph is rerun on those names only. If a node is deleted from the graph, all names declared in those deleted nodes are deleted from every node in the graph, then propagation is rerun on the entire graph but only with those names (as they might be defined in the remaining nodes). While propagation is $O(bn^3)$, with a little bit of thought, it is possible to reduce b and n to make the time more tractable in practice.

6.5 Language Experiences

6.5.1 Cool

Cool is a language designed for teaching programming language design and compiler implementation at the University of California, Berkeley [3]. It is object-oriented, statically typed, and has automatic memory management. Cool is an expression language; all statements have values (thus they also have computable types). Cool programs consist of classes, methods and fields. A program begins by executing the "main" method from the "Main" class, and proceeds until it has returned an integer value. Access permissions are standardized – all classes and methods are public, while all fields are protected. Especially useful for a compiler course, Cool is a small language (contains only five built-in types) and employs a whole-program compilation strategy.

Graph Construction

The language-specific portion of the IG for Cool consists of four node types, six visibility classes, and seven kinds. There are nine graph constructs defined for Cool: the Cool program, each class, each method and field, each let and let binding, each type case and case declaration, and each anonymous inner scope.

The initialization of the IG for Cool begins by initializing the visibility classes. The seven visibility classes are **CLASS**, **METHOD**, **PARAMETER**, **VARIABLE**, **SELF_TYPE**, and **NO_VISIBILITY**. The first four are used to describe the entities that they are named for;
SELF_TYPE is used to spread the association between the keyword **self** and its type in a particular class which is the class itself. **NO_VISIBILITY** is used to label edges that are semantically significant to the structure of the graph, but not semantically significant to the propagation of any names in the graph.

The graph is rooted at the program node. All classes are attached to this node in a star pattern as shown in Figure 6.10. Since all class names are public and visible to all other classes, the edges in this part of the graph are labeled with **CLASS**.

A Cool class is represented by three IG nodes: a class node, a fields hub to which all fields are attached in a linear doubly-linked list, and a methods hub to which all methods are attached in a star pattern. This can be seen in Figure 6.11. Within a class, all class names are visible, method names from the class itself, as well as the binding for SELF_TYPE, a special type that is usually the type of the class itself. The edge from the class to the fields hub is labeled with **CLASS**, **METHOD**

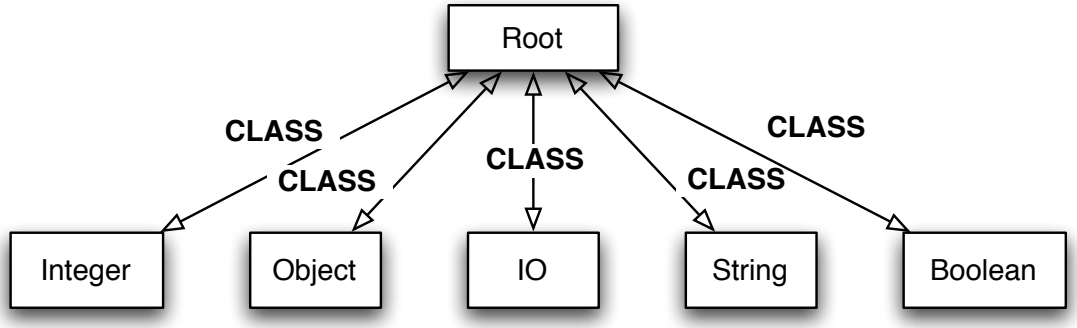


Figure 6.10: Inheritance Graph for the root of a Cool program. There are five built-in classes: Object, IO, String, Boolean, and Integer.

and **SELF_TYPE**. The edge back from the fields hub to the class is labeled only with **VARIABLE** because only variables may be defined within the fields of the class. The edge to the methods hub is labeled with **CLASS**, **VARIABLE** and **SELF_TYPE**. The edge from the methods hub is labeled only with **METHOD** to allow the methods defined in the class to be visible to the fields. Within the class node itself, there are two bindings defined, one mapping the name of the class to its type, and the other mapping the keyword `SELF_TYPE` to the class type with a label of **SELF_TYPE**.

If a class has a super class, it is connected to it by a binding mapping the keyword `SUPER_TYPE` to the super class type. This binding has a label of **NO_VISIBILITY** to indicate that this binding will not propagate anywhere. The super class' method hub is connected to its subclass' method hub with a **METHOD** label. Likewise, the super class' field hub is connected to its subclass' field hub with a **VARIABLE** label. This lets the super class variable and methods flow into its subclass, but not allow any reverse flow.

A method is represented as a single node in the IG. It is connected to the method hub via two edges. The edge from the hub to the method is labeled **CLASS**, **METHOD**, **VARIABLE** and **SELF_TYPE**. The opposite edge is labeled only by **METHOD**. A method node contains local bindings for each of its formal parameters. Each of these parameter bindings is labeled with **PARAMETER**. Finally, a method contains a single binding for its own name, labeled with **METHOD**.

A field is represented as a single node in the IG as well. Fields are linked into the IG via a linear doubly-linked list beginning at the fields hub. They are linked in the order in which they are defined in the source code. At runtime, Cool fields are all initialized to default values determined

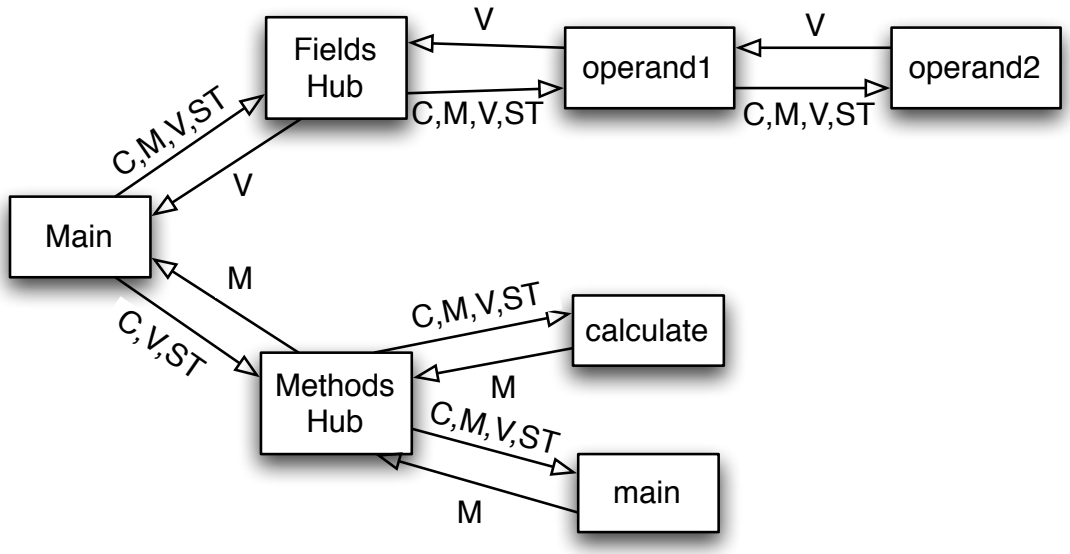


Figure 6.11: Inheritance Graph for a Cool class named Main, with two fields named operand1 and operand2, and two methods named calculate and main. Notice how the fields are attached in a linked list pattern, while the methods are attached in star pattern. The linked list preserves the textual ordering of the fields in the file, and allows an operational semantics analysis to initialize the fields in the proper order. Abbreviations are used for the visibility classes: C → `CLASS`, M → `METHOD`, V → `VARIABLE`, and ST → `SELF_TYPE`.

by their static type, and then reinitialized by their declared initialization code in the order they are declared. There is a pointer from the fields hub to the last field defined. The new field is linked in there via two edges. The edge down from the start of the linked list is labeled **CLASS**, **METHOD**, **VARIABLE** and **SELF_TYPE** to allow these values to flow into the field initializer. The return edge is declared only with **VAR** to let the newly defined field to migrate back up the linked list to the rest of the program.

An anonymous inner scope inherits all names from an outer scope. This kind of scope is also used for the top-level method body. It is connected to the outer scope by a downward edge (from the outer scope to the inner scope) labeled with **CLASS**, **METHOD**, **VARIABLE**, **PARAMETER** and **SELF_TYPE**. There is no return edge because names cannot escape an inner scope in Cool.

`Let` and `typecase` are the only two constructs within a method body in which new names may be defined. Each consists of a pair of IG nodes: one for the construct itself, and one for each binding defined within. If either construct declares more than one binding, it is equivalent to

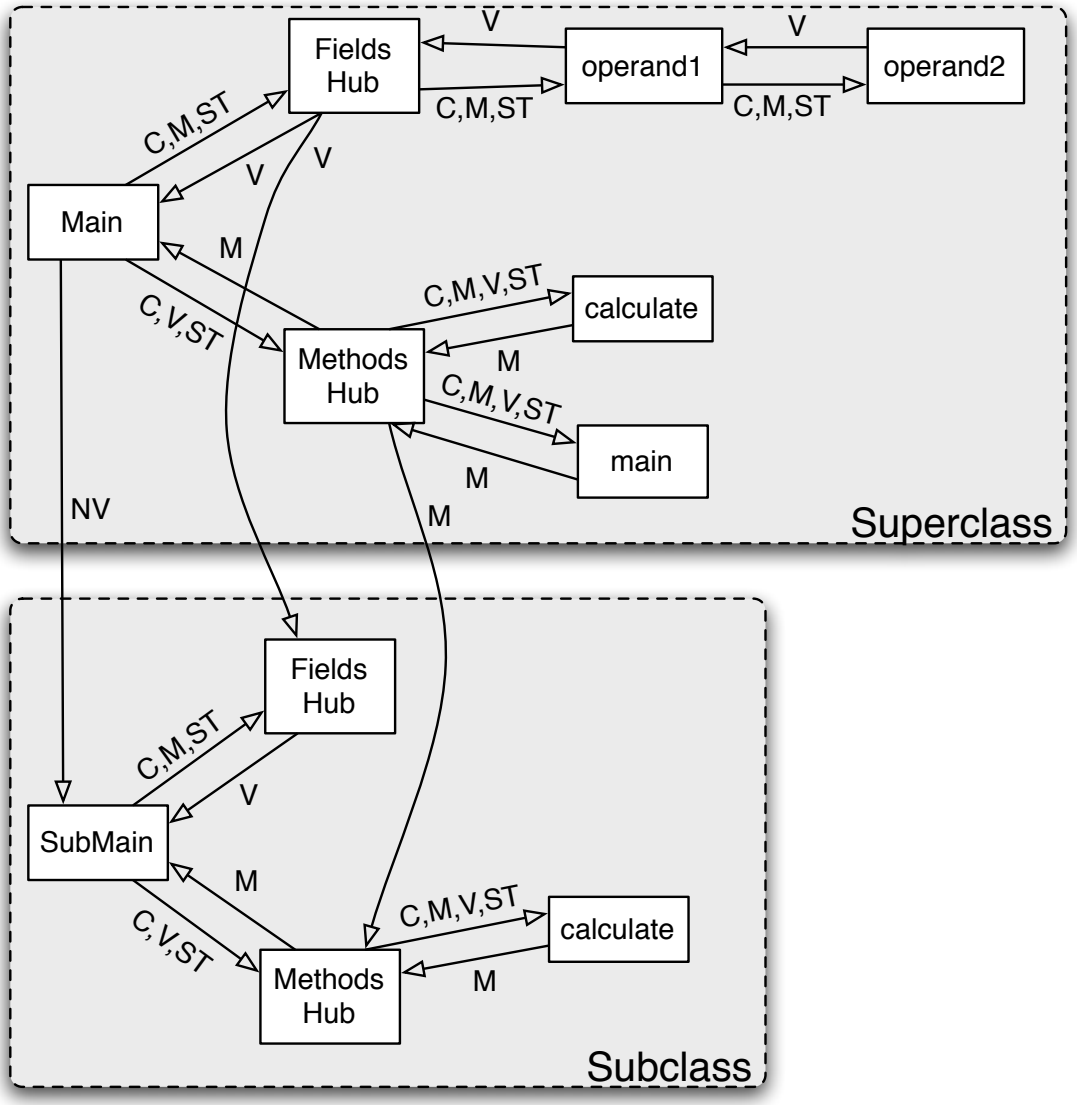


Figure 6.12: Inheritance Graph for two Cool classes in a superclass–subclass relationship. Subclass SubMain overrides method calculate from its superclass Main. The superclass' field hub and method hub are connected to the subclass' hubs to allow field and method definitions to flow from superclass to subclass. The class nodes themselves are connected, but no bindings will flow over that edge. Abbreviations are used for the visibility classes: C → `CLASS`, M → `METHOD`, V → `VARIABLE`, and ST → `SELF_TYPE`.

having each binding defined one at a time in the order it appears within the source code. Let and typecase nodes are connected from their outer scope node by the same bindings as for anonymous

```
calculate(x : Integer, y : Integer) : int {
   let x : Integer <- 6 in
      y + (let z : Object in
             x * operand1)
}
```

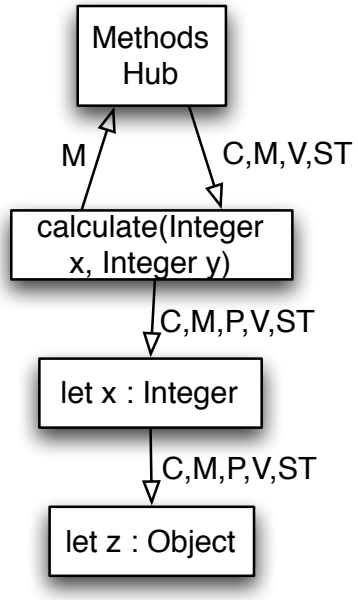


Figure 6.13: Inheritance Graph for a method named calculate. Calculate takes two parameters, x and y and has two inner scopes. The first inner scope overrides x with a new value. The second inner scope introduces a new variable z. Notice how all bindings in the method body flow down into the body, but not back up. Abbreviations are used for the visibility classes: C → `CLASS`, M → `METHOD`, V → `VARIABLE`, ST → `SELF_TYPE`, and P → `PARAMETER`.

inner scopes. Their bindings are are connected in the same way. Each binding maps the new name to the type of the variable declared in the let or typecase expression. Each of these bindings is labeled with **VARIABLE**, enabling them to flow downward into inner scopes, but not back up.

An example of a method body is shown in Figure 6.13. The `calculate` method is connected to the method hub. It has two parameters, `x` and `y`, both of which are Integers. The parameters flow down into the body of the method via edges labeled with visibility class **PARAMETER**. The first `let` statement overrides the definition of `x`, and the second `let` statement introduces a new variable `z`. Each variable flows only downward through the method body due to the labeling of visibility classes on the edges.

Table 6.1 below describes the results of a clash between Binding A and Binding B. Classes are all defined at top-level and may not be duplicated. To enable type-checking to proceed anyway, only a warning is indicated. There is no method overloading in Cool, thus methods with the same name but a different method signature are disallowed. Methods may be overridden in a subclass, however, if they have the same name and signature. Thus, the local binding of the method in the subclass wins over the propagated binding from the superclass. Cool does not allow two fields (or method parameters) to be defined with the same name, regardless of type. However, all variables may be shadowed by a variable with the same name (irrespective of type) defined in an inner scope.

	Propagated Binding (P)					
Local Binding (L)	Class	Method	Field	Parameter	Let	Typecase
Class	P (Warning)	N/A	N/A	N/A	N/A	N/A
Method	N/A	L (Error if diff signature)	N/A	N/A	N/A	N/A
Field	N/A	N/A	Error	L	L	L
Parameter	N/A	N/A	L	Error	L	L
Let	N/A	N/A	L	L	L	L
Typecase	N/A	N/A	L	L	L	L

Table 6.1: Clash Table for Cool Inheritance Graph. Each cell indicates which binding wins when both reach the same IG node during propagation.

When propagation has finished, each IG node contains every name visible in its associated scope in the program. Any errors caused by duplicate names are flagged during propagation. Thus, typechecking can concern itself solely with proving type safety without worrying about duplicate name detection. Typechecking in Cool begins constructing the class inheritance hierarchy and detecting any illegal cycles. Connectivity testing in the class hierarchy is not necessary because any class without a declared superclass is automatically declared to subclass Object. Next comes a recursive-descent tree walk through the bodies of each method and initializers of each field in every class defined in the program. Each use of a variable is looked up in the appropriate IG node for the variable's type. If the type employed does not conform to its use, an error is indicated.

The Cool language specification contains tags to support incremental IG update, as described in Section 6.4. This update specification is used to minimize edits to the IG when only small portions of the Cool program have changed. Since most Cool programs are small, this does not appreciably improve performance, but it does provide excellent high-level change information, which for human studies of programmers is eminently suitable for coding the trace of a programmer's editing session.

6.6 Applications

The IG provides a persistent, incrementally updateable data structure for names and entities declared in a program. It can answer queries of the form, "what does this name mean in this program location?" and "what names are visible in this program location?" It can also answer the queries, "is this entity visible in this program location?" and "what is the simplest name of this entity in this program location?" The answers to these queries form the basis of several useful and common applications.

6.6.1 Name Lookup and Type Checking In Java

Type checking ensures that all uses of names correspond to valid operations according to the language's type system. Type checking often takes the form of a tree walk, computing the type of each name and expression and validating its correct application. The inherent question asked during type checking is "what does this name mean at this program position?" which is the main question that the IG was designed to answer.

Name lookup is required for type checking. In many languages, all names that are visible in a given scope will have propagated to that scope's IG node, enabling a constant time name lookup. Java and C++, however, have complex name lookup rules which are not amenable to this kind of propagation.

In Java, looking up the single (unqualified) name of a method requires looking in the current class for methods of the same name. The methods defined in all superclasses and superinterfaces of the class are checked next, followed by the enclosing class and its superclasses, and so forth. If the single name denotes a package, class, interface, field or local variable name, the search is similar, except that the scope in which the reference occurs is searched first, and enclosing methods are added to the search path. If the name is still not found, the compilation unit and imported classes are searched, followed by the current package, imported packages, and then the top-level package hierarchy.

Qualified names are looked up one single name at a time in the order in which they appear. The first single name is resolved to an IG node as described above. The rest of the names are resolved by looking only in the preceding IG node and its superclass or superinterface hierarchy, but *not* anywhere else.

The kind of reference can affect the search. If a single name is used as the type in a type declaration, then it cannot be a field or variable name. An example is the variable declaration `Foo`

`f;` Foo must be a type. If there is a Foo that is a variable, it is ignored. The Java name lookup rules state that any time during a name lookup, if a field name is found with the same name as a package, class or interface name, the field name is ignored. However, if the reference is ambiguous (*e.g.* a variable used in a math expression), the field name must be considered, and in fact, if inherited by the local scope, beats the package, class or interface name. An example of this case is the statement `f = Foo;` Foo could be a type, field or variable; if Foo is defined as a variable and as a type, Foo is seen as a variable, not a type. If a qualified name is used as a variable in an expression, the final single name in the qualified name must be a field or variable. This situation is the same as the one above, the name is preferred as a variable, rather than a type.

Several problems were not anticipated at the time the IG model was designed (nor are they handled by any other name resolution model). First, in the era when the IG was designed, it was possible to completely separate name resolution from type checking in all programming languages. With C++ and Java, this is no longer the case. Superclasses cannot be resolved without name resolution and graph construction being complete. If superclasses were connected to subclasses by IG edges, the graph would change structurally every time a superclass was resolved, requiring repropagation and reexecution of name resolution. In addition, the inheritance of names from superclasses and outer classes must occur in a prescribed order, which would require ordered processing of VC-labeled edges, whereas inheritance of bindings via edges in the graph is unordered. There is no way to express such ordering rules in the IG. Consequently, the IG we designed for Java does not contain edges connecting superclasses to subclasses or outer classes to inner classes. Third, some names existing in the same namespace (*e.g.* package, class, interface, field and local variable names) are not represented in the *CLASH()* function for Java. This is because context-sensitive name lookup cannot be easily solved without inserting superclass and outer class IG edges.

6.6.2 Spoken Program Disambiguation

Using programming by voice in a program editor, programmers incrementally modify and add to programs using speech. Since each utterance occurs in the context of an existing program, program context can be used to disambiguate the parse forest and choose the most appropriate interpretation of the input.

There are two ways to use the IG to disambiguate the input, either rule out each interpretation based on the legality of individual names found within, or rule out the entire interpretation simply by type checking it. Both options are presented here.

First, the interpretations of the ambiguous parse forest are enumerated. For a given utterance, there can be quite a few interpretations, and many of them can have up to 30 to 40 words inside, each of which needs to be validated. It is vital to have quick name lookups for all possible names at a program point. Each interpretation is a small chunk of code that is locally syntactically correct, but whose words inside must be validated against the program. For example, a user who says "foo bar" may have meant `foo(bar)` or `foo.bar`. Using the IG, one can look up the identifier `foo` and find out its kind and type. If `foo` is not a function definition, the first interpretation can be rejected. If `foo` is not a field of the current class, or the name of a class with a static field `bar` or the name of a package containing a class name `bar`, the second option can be rejected. Each interpretation is analyzed until all but one is rejected, leaving the most likely interpretation for validation by the programmer.

In the other disambiguation option, the IG is used as if the interpretations are actual edits to the program. Each interpretation is inserted into the graph at a location appropriate to its context. For example, an edit may have added a new statement in the middle of a list of statements in a method. The appropriate anchoring IG node is found and used to insert any IG nodes the new interpretation may have created. The program is then repropagated and type checked. If any errors appear (that were not in the original program) the interpretation is not legal.

A key advantage of the IG over other name resolution formalisms is the persistence of information. This persistence enables disambiguation of spoken input at any point in the program without recomputation of symbol table information.

6.6.3 Eclipse

The implementation of the IG presented here is part of the Harmonia program analysis framework. Harmonia has been plugged into the Eclipse IDE [23] and programmed to extend the Java text editor. Eclipse provides many time-saving semantics-based features to Java programmers including code completion, Javadoc text hovers, compiler error display, use-def hyperlinks, and semantic-based search. All of these are easily implementable using the Inheritance Graph. We have implemented several of them as a test of the power of the Harmonia framework.

Code completion is the display of the results of the query, "what are the names visible in this list of program scopes?" The list of the program scopes is gathered from asking "what are the meanings of this identifier at this program position?" and resolving the resulting entities to their defining program scope. This list is easily filtered based on the prefix of any words further typed

by the programmer. Javadoc text hovers are the result of a similar query, "what are the meanings of this name as a (method/field/class/package) in this program position?" The Javadoc of the entity returned is then displayed in a text hover. Compiler errors are computed during propagation and type checking phases of program analysis and attached to parse tree nodes for display by the program editor. Use-def hyperlinks are created for each name in the program by asking "what is the meaning of this name at this program point?" and resolving the resulting entity to its defining node. Semantics-based search involves some preprocessing. After semantic analysis occurs, tuples of names, kinds, types and defining nodes are gathered from the IG for a given program and stored in hashtables for later speedy access by a user search on any stored attribute.

6.7 Implementation

The IG data structure and algorithms are implemented in C++. The language-independent portion is a C++ template, parameterized by the Name and Entity types and the number of visibility classes. The language-dependent portion is the instantiation of the template, defining the Name and Entity types, the visibility classes used, and the implementations of the *CLASH()* and *RESOLVE-CLASH()* functions. In addition, the language implementor designs the graph structure: the kinds of nodes that are required, how they are connected to one another, what the different visibility classes mean and how they are placed on bindings and edges to achieve the desired name visibility. We have found that there are several graph design patterns that can achieve many of the name visibility rules described in Section 6.1 and have pointed these out wherever appropriate in the Java Example in Section 6.2. Several diagnostic functions are provided in the implementation to print out the graph in text and dot [25] file formats. These are invaluable for debugging incorrect graph structures. In addition, the implementation contains several cross-checking validation functions to ensure the graph is connected and that bindings propagate where they should.

6.7.1 Performance

Scalability is an important component of program analysis. We profiled the IG on the Java 1.4.2 library (4,137 files, 6,705 classes and interfaces, 1,293 kLOC) for memory consumption and speed, even though it has been optimized for neither. The IG for the source code of the Java library (should the entire library be loaded into memory at once) consists of 582,756 nodes (and 496,116 bindings), for a total memory size of 98 MB. The IG for just the class files of the Java

library consists of 423,281 nodes, for a total memory size of 63 MB. To understand this in context, the uncompressed size of all of the Java 1.4.2 library class files is 32 MB on disk. The IG size is within a constant factor of the Java 1.4.2 library size, and is easily tractable on today's typical one-gigabyte development machines. Note that almost all Java applications developed today are dwarfed in size by the Java library itself.

We calculated that the IG size in nodes, bindings, and bytes scales linearly with lines of code. There are approximately 450 IG nodes per kLOC. There are 300 local bindings per kLOC. Bindings are not evenly distributed through the nodes. In the Java library there is an average of 120 local bindings per file (standard deviation of 172), but a few files have many more local bindings (the maximum is 2,430 local bindings).

We have made design changes to the Java IG to improve asymptotic performance. Each file is loaded into an IG subgraph which is connected to the IG only at its CompilationUnit IG node. There is no flow of names from outside the CompilationUnit into the file's subgraph. When changes are made to the contents of the file, it can be reanalyzed in isolation. Since IG propagation is $O(bn^3)$, lowering the number of nodes in this way has a significant effect on propagation time. Another change is to load portions of the IG on-demand. As each file (especially those in the Java 1.4.2 library) is referenced by a name in a user program, it can be loaded into the IG, propagated independently, and have its names be available to type checking and other static analyses.

6.8 Related Work

Graham, Joy, and Roubine introduced a technique to handle languages with a mix of open and closed scopes [35]. They showed that by numbering each scope by its lexical distance from the global scope, bindings could be marked by numbers indicating in which scopes they were visible. They described two methods for implementing qualified names and opening of closed scopes. One was to take each single name in a qualified name and place it in the main symbol table. The second method was to maintain a secondary table of the qualified names which was searched during a name lookup. While they chose the first method, our IG data structure employs the second. As this paper was written in 1979, it focuses heavily on memory footprint and speed, which are not nearly as important as they once were. Our approach sacrifices some speed and memory for more generality.

Reiss uses a language-independent specification for a symbol table, with names, objects (what we call entities), and scopes [83]. The paper employs a two-staged name lookup procedure during lexing and parsing that returns a name given a lexeme, and an entity given a name. The

IG performs both stages during semantic analysis. Reiss' name and object classes are similar to our notion of "kind." Reiss' names are divided into explicit classes, but our kinds are implicitly assigned. Reiss' work does not handle separate namespaces for identifiers, does not support non-lexical entities, and has no facility for non-source-code-based reference bindings that link entities in the program.

Klug described a declarative name resolution system [51] which was subsequently replaced by an imperative procedural specification [52] similar in form to a nested set of symbol tables. It was unclear from the papers how the specification should be applied to an actual programming language, how it should be implemented, or its performance. Klug continued by describing a collection of visibility constructs that programming languages could exhibit, but the paper lacked many details [53].

Vorthmann's Visibility Networks [102, 103] come closest to the IG in design. Vorthmann's graphical representation of the network illustrates concisely how the system will behave. The *CLASH()* function has been simplified to three operators (!, = and ↑) applied to edges, corresponding to error, reference and shadow clash resolutions. If two in-edges to a node are labeled with !, none of their bindings can clash or there is an error. If two in-edges are labeled with =, two bindings can clash, but they must be equal in value. If an in-edge is labeled with ↑, its bindings beat others in a clash. This simplified approach can handle many languages, but is not powerful enough to handle Java or C++ due to their mix of lexical scoping and object-oriented inheritance. In addition, Vorthmann's clash operators do not take entity kind into account, leading to an awkward specification of clashes of names that are not in the same namespace as one another. Visibility Networks are also not powerful enough to describe dynamic scoping or other runtime properties, which are describable in the IG.

There are many other automated approaches to generated name resolution semantic analyses. One notable example is the attribute grammar. Many languages have been described using attribute grammars alone. Despite having a long history, attribute grammars have not achieved the same success as other automated program analysis generators such as lexer and parser generation.

6.9 Future Work

The IG is implemented as a single connected graph which must be entirely in memory at all times. However, in large, hierarchically organized languages like Java and C#, most of the language and its libraries are unused by many programs and thus unneeded in the IG data structure.

While the set of referenced classes cannot be anticipated before the program is written, classes that are infrequently used can be flushed to disk with an MRU paging strategy. Our Java IG structure is carefully compartmentalized to inhibit name propagation across files, which makes the paging approach possible.

All languages support compilation of multiple program files, merging the resulting binaries into a single executable program. When programmers edit these multiple source code files in a program editor, they make changes to the names declared in the file that are used by other files. Our implementation of the IG does not track these def-use dependencies when deciding when to rerun the type checker and other static semantic analyses. A naive coarse-grained strategy can be employed which reruns all analyses after any change to a name, but this can have serious performance implications. A better approach would be to track dependencies on a compilation unit or program entity level so that only the uses of the exported name will have its analyses rerun.

Chapter 7

SPEED: SPEech EDitor

The most tangible artifact of our research is SPEED, the SPEech EDitor. SPEED is a Java program editor embedded in the Eclipse IDE [23]. Eclipse is a Java-based IDE mainly used for developing Java software. It is designed to be extensible, and is fortunately open-source, for when the implementation does not live up to the design goals. SPEED uses the Eclipse IDE both to maintain the program code being written and to support our voice-based user interface innovations.

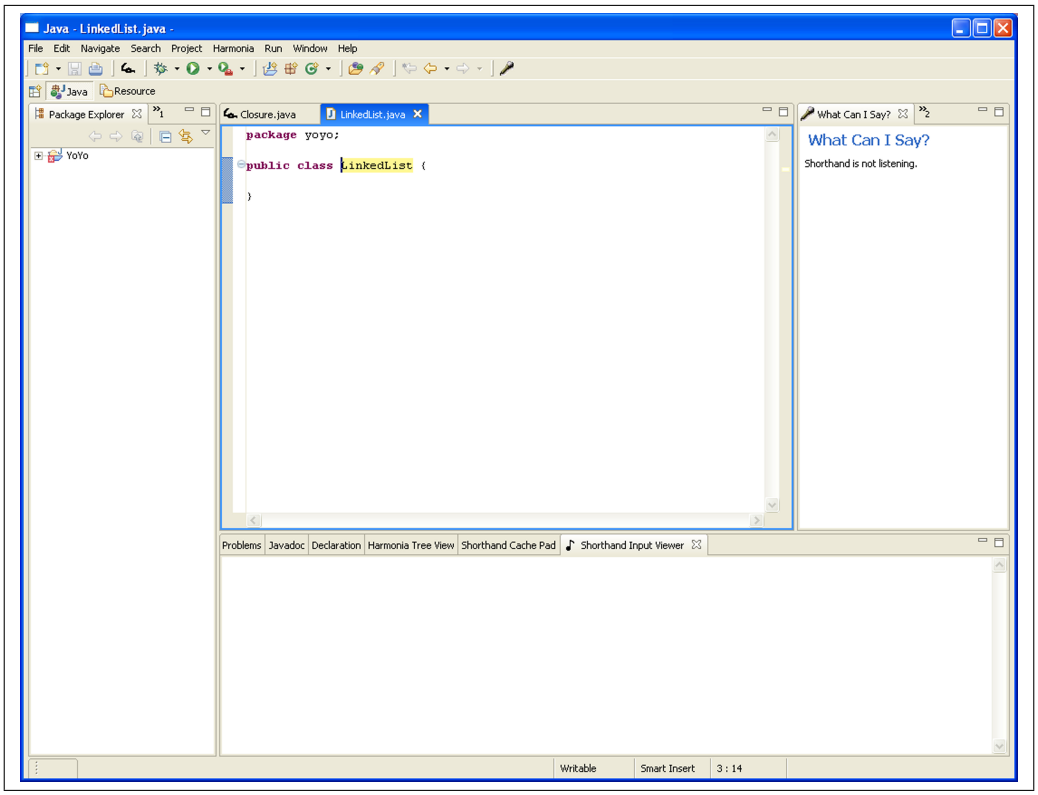
SPEED uses Nuance's Dragon NaturallySpeaking [69] to listen to the user, and the analysis systems described in the previous chapters to understand what the user said. Its basic code authoring model is to have the programmer speak fragments of code or names in Spoken Java, translate the results into Java, and insert them into the program. For code editing, the model is similar. The programmer selects a Java construct to edit. The system translates it into Spoken Java, lets the user edit it via speech, and then translates the edited version back into Java. Since Spoken Java to Java translation can produce multiple answers, the user is presented with a list of translations to choose the one he prefers.

This chapter consists of two sections. The first section walks the reader through a programming scenario using SPEED. The second section describes the major components of SPEED and how they work. Chapter 8 will discuss our user studies exploring the utility of SPEED with professional Java programmers.

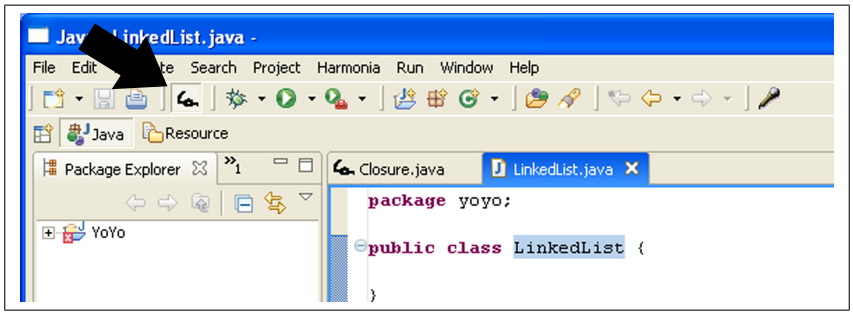
7.1 Sample Workflow

We introduce Shari, a programmer experienced in using voice recognition, experienced in Java, and experienced using the Eclipse programming environment, who wishes to use SPEED to

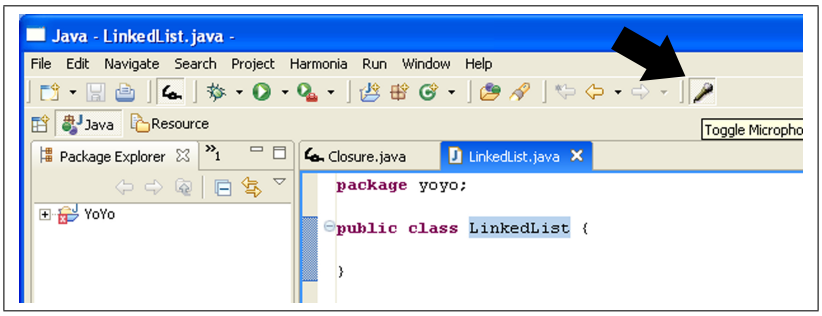
create a linked list data structure for primitive integer values. Shari begins by bringing up Eclipse and creating a new class within her project called LinkedList.



She then opens the editor for LinkedList and turns on Shorthand by clicking on the Shorthand button in the Eclipse toolbar.



She then puts on her headset microphone and activates it by clicking on the microphone button in the Eclipse toolbar.

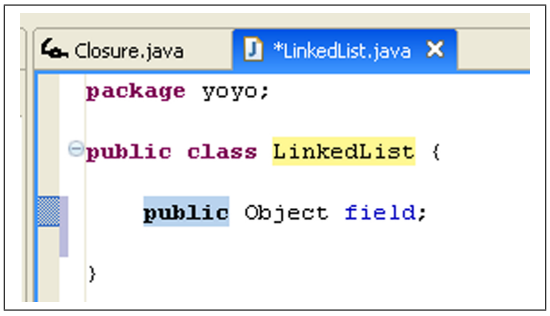


When Eclipse creates a class, it adds a package declaration and an empty class declaration. Linked lists have two fields, one of the element itself which will be of type primitive integer, and the other a pointer to the next element in the list. She begins by navigating to the inside of the class body and inserting a field.

"Jump to class linked list." (absolute name-oriented navigation)

"Go down." (cursor based navigation)

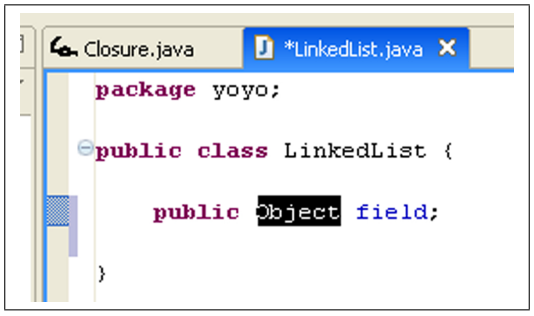
"Insert field." (example of code template insertion)



The field appears with a default type `Object` and default name `field`. Now she must fix the type and name of the field.

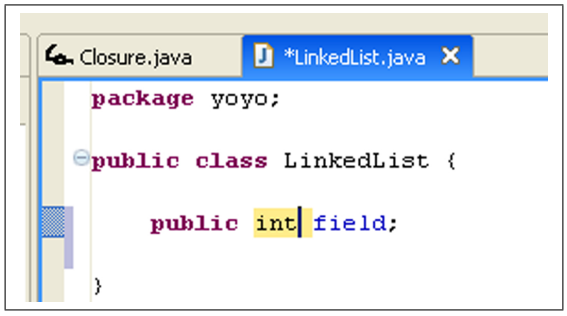
"Go right."

"Edit this."



The edit operation translates the currently selected item into Spoken Java, and then waits for dictation-style input.

“int. Done.”



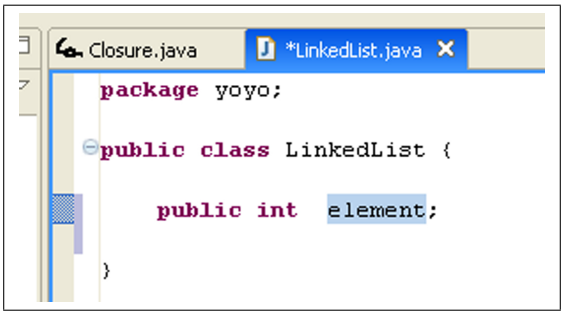
She speaks the type “int” and then “done” in order to indicate that she is finished entering the code. SPEED then translates her utterance back into Java. Since there is only one interpretation: “int” SPEED inserts it automatically.

“Go right.”

“Edit this.”

“element. Done.”

These utterances edit the name of the field and change it from “field” to “element”.



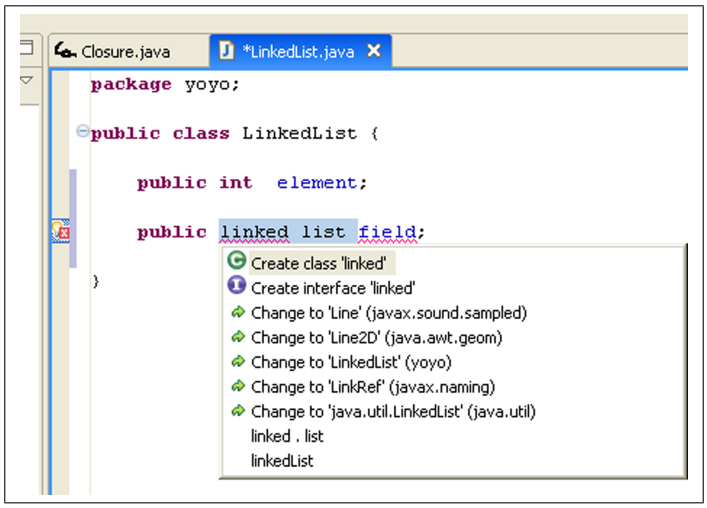
She repeats the preceding utterances to insert another field of type “LinkedList”.

“Insert field.”

“Go right.”

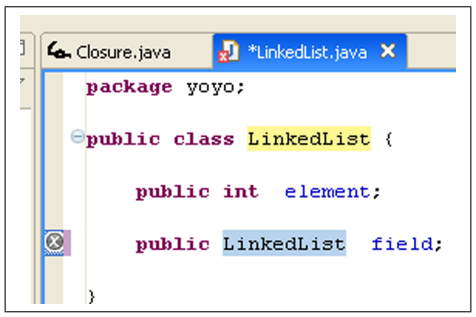
“Edit code.”

“linked list. Done.”



As uttered by a person and interpreted by a voice recognizer, LinkedList comes out as two words separated by space. When Shari says "done," a translation popup menu appears listing the possible context-limited interpretations of those two words. There are two choices for the type name here. The first is `linked.list`, a reference of an inner class `list` in outer class `linked`. The other, the choice Shari prefers, has concatenated the two words together to form a single type name, `linkedList`. In this version of SPEED, Shari must click on her choice to enter it into the system. However, in a production quality version of SPEED, the choices would be numbered, and Shari would choose by speaking the number of the desired menu item. Note how the translator changed the capitalization of the words to CamelCase, the standard style for Java identifiers.

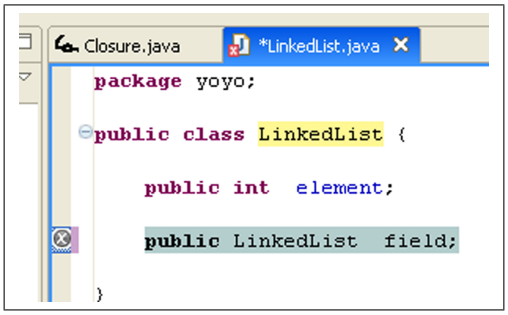
Since the type name `linkedList` should be capitalized, Shari says:
"Cap that."



Now she must insert a constructor to set the fields when the linked list is instantiated. She could say "insert constructor" to insert a constructor template. Another way to do it is to simply speak the natural language words in the constructor.

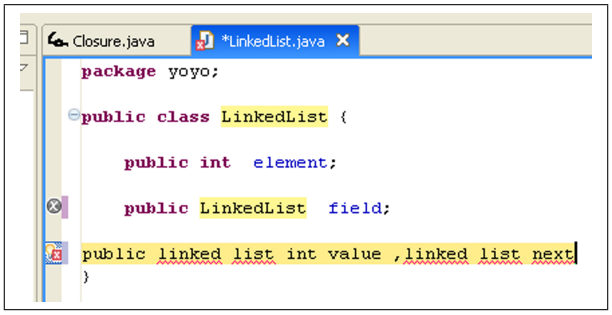
In order to insert a new constructor element, Shari must select the second field and say:

“Insert after.”



She then speaks the literal words in the LinkedList constructor.

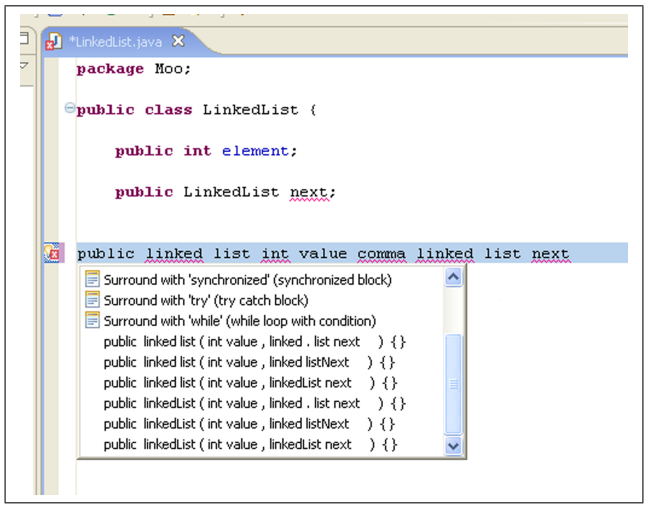
“public linked list int value comma linked list next. Done.”



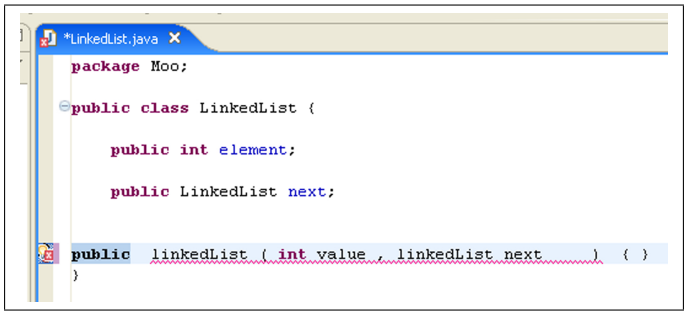
Notice the lack of punctuation, spelling, and lack of control of the spaces between words. When this is translated to Java, there are six choices from which to choose.

```
public linked list (int value, linked.list next) {}
public linked list (int value, linked listNext)  {}
public linked list (int value, linkedList next)  {}
public linkedList  (int value, linked.list next) {}
public linkedList  (int value, linked listNext)  {}
public linkedList  (int value, linkedList next)  {}
```

Shari knows from experience that she must not forget the comma! Without it there are 2,654,208 possible interpretations. Spoken Java can be very ambiguous without any punctuation.



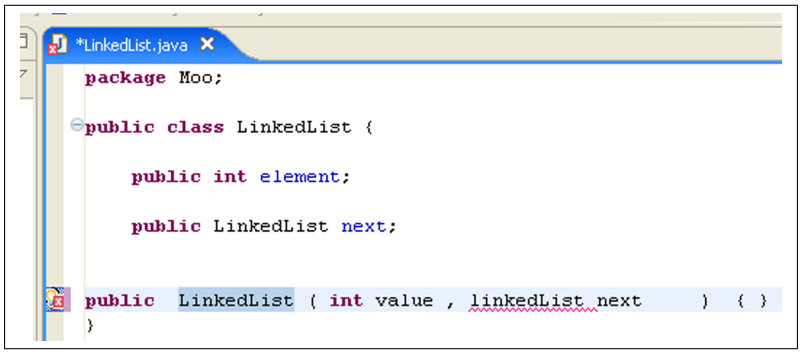
Shari chooses `public linkedList(int value, linkedList next) {}`.



The spelling of linkedList is missing its capital letter. In this prototype version of SPEED, the capitalization of identifiers is not yet constrained to names already in scope (such as the `LinkedList` class name). She navigates to the name of the constructor and capitalizes it:

“Jump to constructor linked list.”

“Cap that.”

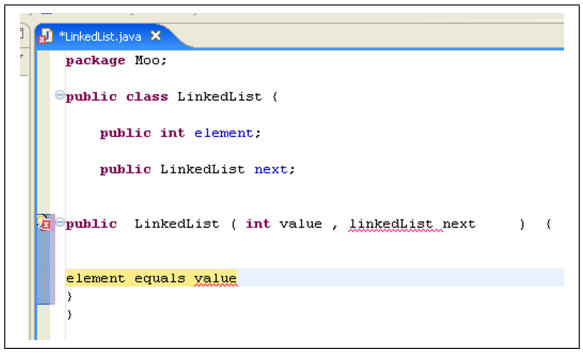


She navigates into the body of the constructor and assigns the values of the parameters to each field.

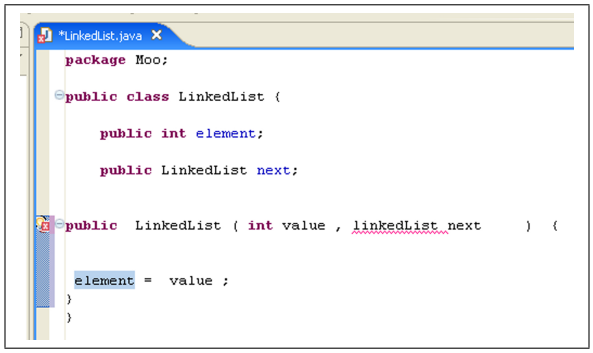
“Go down.”

“Insert here.”

“element equals value. Done.”



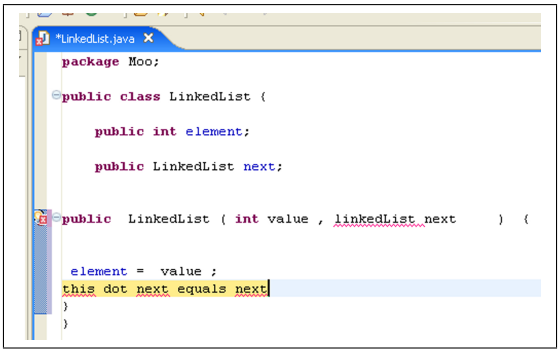
This only has one interpretation, so the translate popup menu does not appear.



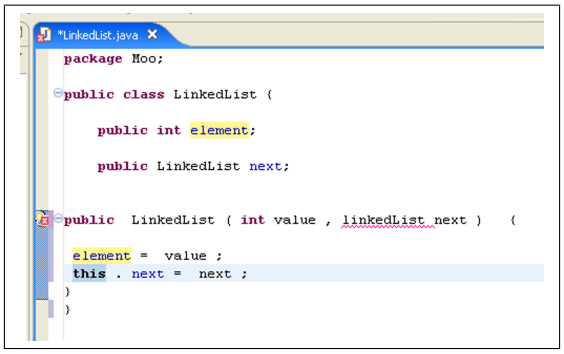
“Expand selection. Expand Selection.”

“Insert after.”

“this dot next equals next. Done.”



This translates to `this.next = next;`. There is only one interpretation. Even if we had left out the dot, there still would have been only one translation.

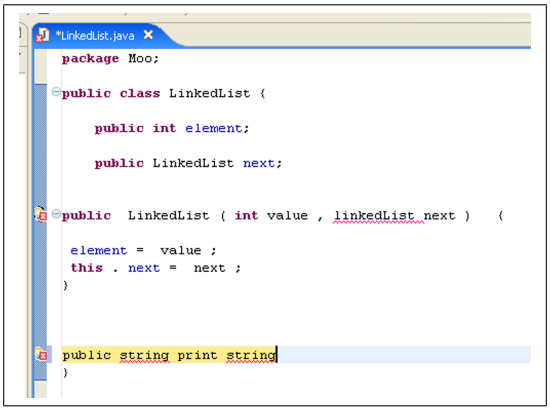


Next, Shari creates a `printString` method, to print out each list element.

"Go down."

"Insert after."

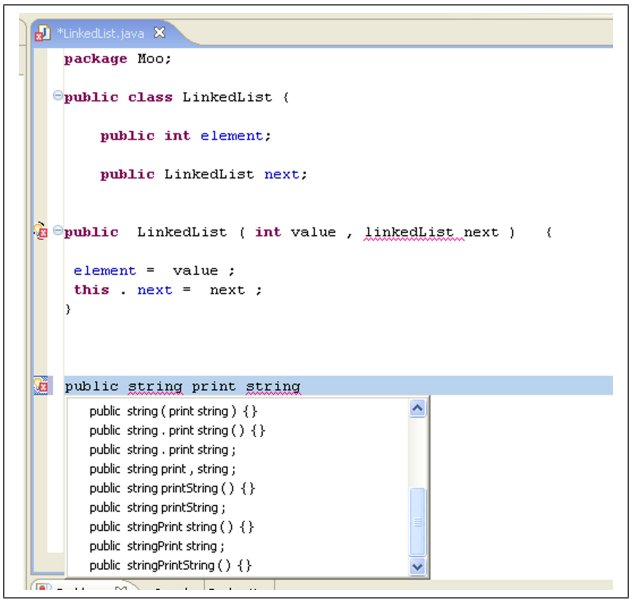
"public string print string. Done."



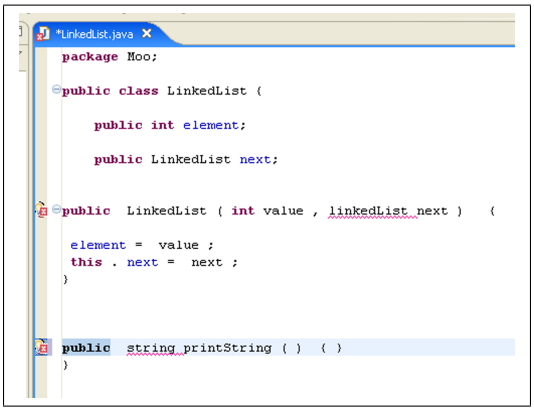
There are 11 possible interpretations of even this simple method declaration.

```
public string() { print(string); }
public string() { print string; }
public string(print string ) {}
public string.print string() {}
public string.print string;
public string print, string;
public string printString() {}
public string printString;
public stringPrint string() {}
public stringPrint string;
public stringPrintString() {}
```

They appear in the translation popup menu in SPEED.

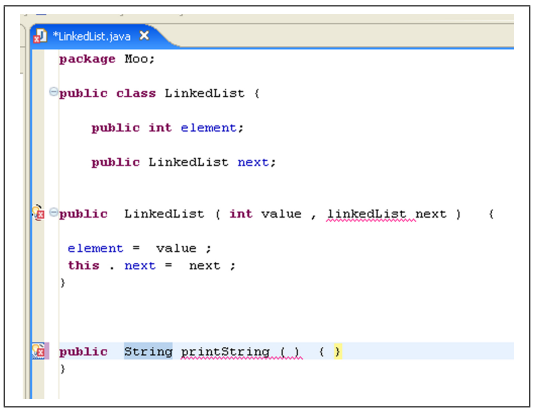


Shari chooses `public string printString() {}`.



“Go right.”

“Cap that.”

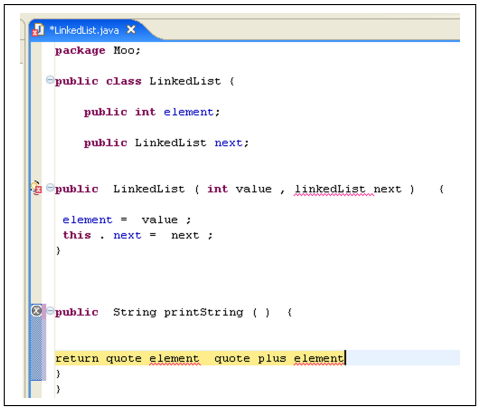


“Go right.”

“Go right.”

“Insert here.”

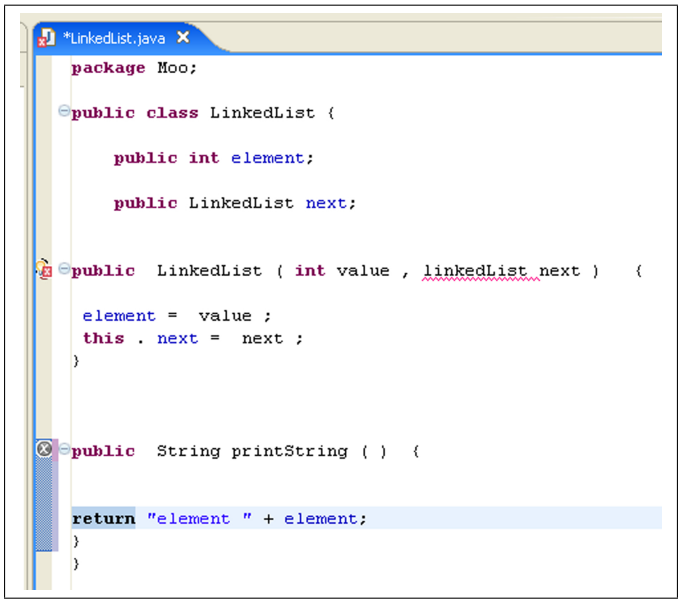
“return quote element space quote plus element. Done”



The final program is as follows:

```
package Moo;

public class LinkedList {

   public int element;

   public LinkedList next;

   public LinkedList(int value, LinkedList next) {
      element = value; this.next = next;
   }
```

```
    public String toString() {
        return "element " + element;
    }
}
```



7.2 Early SPEED Prototypes

Prior to creating SPEED, we created two prototype editors that integrated speech recognition with XEmacs. Neither the syntax analysis nor the semantic analysis were developed at the time, so we added code template expansion to see how far one could get. It turns out that it is not far. In fact, while structures were easy to insert into the editor, the vast majority (around 85%) of words in the Java program were identifiers. Identifiers appeared often, and when they appeared were often contiguous, producing long strings of identifiers. For example, a valid Java statement is `javax.swing.UIManager.setLookAndFeel( javax.swing.UIManager.getSystemLookAndFeelClassName())`. When spoken, there are 21 distinct words that can be uttered in a row. No code template expansion service is going to be able to help very much here.

7.3 Final SPEED Design

We activate SPEED using the commercial speech recognizer Nuance Dragon NaturallySpeaking [69]. Dragon NaturallySpeaking is available only for the Microsoft Windows operating system, thus limiting our tool to running on Windows. Harmonia itself, and all of the other technology described here runs on Windows, Linux, Solaris and MacOS X. We attach Dragon to Eclipse using a third-party Java Speech API (JSAPI) [93] library called Cloudgarden TalkingJava SDK [16]. Cloudgarden's JSAPI implementation works with all Microsoft Speech API-compliant (SAPI-compliant) speech recognizers such as IBM ViaVoice [39], Dragon [69], and Microsoft's own Speech SDK [38].[1]

SPEED is composed of seven main parts: Eclipse's JDT, Shorthand, Speech Recognition plugin, Context-sensitive Mouse Grid, Cache Pad, What Can I Say?, and How Do I Say That?. The next sections will explain the implementation of each of them.

7.3.1 Eclipse JDT

Eclipse has been designed to be an extensible platform for building rich client applications. The star of their application suite is the Eclipse Java Development Toolkit (JDT). The JDT, shown in Figure 7.1, contains a syntax-aware program editor, an outline view showing all members of the current document, and a package hierarchy view showing all packages and classes in the workspace. The JDT provides a language-aware search tool, enabling programmers to look for class names, method names, field names, strings, and other Java-specific entities. Programmers can also visualize the callers of a method, the superclass and superinterfaces and subclass and subinterface for a class or interface. Code completion allows short several letter abbreviations to be expanded into code templates that can be filled in like a Web form. Java-specific refactoring support is extensive, both on the package and class structural level as well as within method bodies. JDT support also extends to Java debugging on par with most other GUI-based IDEs for Java.

The JDT is not merely a passive editor waiting for users to ask for analyses and refactoring support. Behind the scenes, it uses up to seven Java parsers to analyze the code being written for compiler errors, potential runtime errors, code style problems, and other suggestions that it calls

[1] At the time of this dissertation, the desktop speech recognition market has been consolidating. IBM no longer actively develops ViaVoice. Nuance, which purchased and develops the Dragon NaturallySpeaking recognizer also now sells IBM ViaVoice for IBM. Microsoft's Speech SDK has shipped with Windows XP and Microsoft Office, but has not otherwise been marketed very strongly. All other recognizers on the market are intended for backend use supporting phone-based voice recognition for call centers.

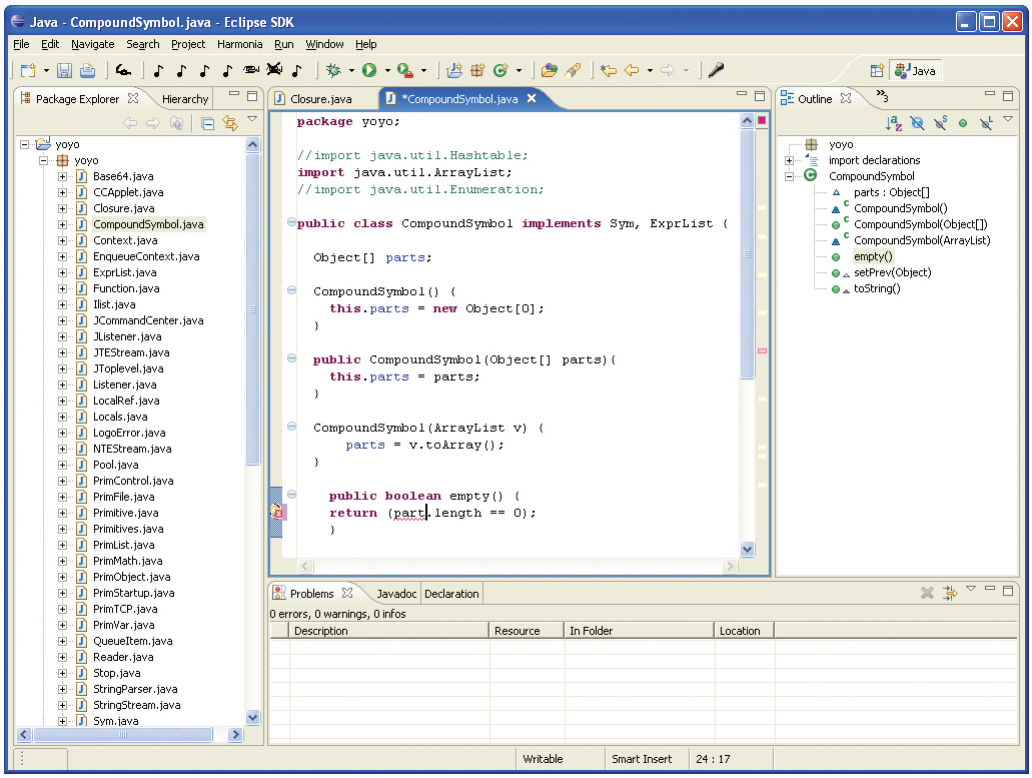


Figure 7.1: A screenshot of the Eclipse Java Development Toolkit editing a file named CompoundSymbol.java. Notice the compiler error indicated by the red squiggly underline under the word "part" in the method empty().

Quick Assists. Quick Assists appear in the left-hand gutter as icons with a light bulb, or as squiggly underlines (red for errors, yellow for suggestions) underneath lines of code. When the user activates a quick assist, a menu of suggestions pops up, requiring only a click to enact his changes on the code.

All of the features provided by the JDT (except the program editor) have been speech-enabled by Jeff Gray's SpeechClipse project at the University of Alabama [90]. He and his colleagues have created extensive speech command grammars for the menus, dialog boxes and the GUI, but did not speech-enable the program editor, which is the concern of this dissertation.

7.3.2 Shorthand

Shorthand is an Eclipse plug-in that we created as part of the Harmonia project. It is designed to enable small sequences of keystrokes to create and modify program code. Programmers can move a structural cursor (a cursor that highlights words, expressions, statements, blocks, meth-

ods, classes, etc) through the program using arrow keys. The cursor can be expanded or shrunk in size to encompass more or less program structure.

Shorthand is built on top of the Eclipse IDE, JDT, and Harmonia program analysis framework. Shorthand extends the JDT CompilationUnitEditor with a editing model akin to structure-based editors. Harmonia analyzes the program text in the background and provides a parse tree representation for Shorthand to use. Shorthand is designed around a "box" model. A "box" is a selectable entity in the program that maps to one or more parse tree nodes. In Shorthand, the current box determines the keystrokes available and the actions they perform. First, the box and its associated node are examined through a set of parse tree patterns loaded from an XML file. For each pattern that matches, a set of *action maps* is activated. Each action map (also loaded from an XML file) defines a set of keystrokes and the actions they invoke. Thus, by activating a set of action maps, a set of keystrokes is activated for the programmer to press. When the box is moved around (via navigation) the set of action maps is updated. Action maps are composable; each set of keystrokes should be mutually exclusive. In the case when this is not possible, there is a priority ordering system used to determine which action's keystroke will win.

Context-sensitive actions are dependent both on structural and temporal state. For example, selecting the `public` keyword in a class declaration enables the "p modifier" action. If the user presses "p" on the keyboard, Shorthand will change `public` to `protected`. If `protected` were highlighted instead, "p" would change it to `private`. Pressing "p" one more time would cycle back to `public`. Other commands are available to insert code templates (class, interface, method, constructor and field), or sequences of entities (e.g. new statement, new expression, new variable). The most powerful action is the activated box. When "spacebar" is pressed, the current selection activates, meaning that it becomes editable by free form text (as if the user were programming in a plain text editor). The boundaries of the activated box are fixed; even if all the text is deleted, the box is still activated and editable. The activated box can be committed or aborted when the programmer is done. When aborted, the original text is returned to the document. If committed, the new text replaces the old.

An action may edit code or navigate through the program. Navigation simply moves the box around to a particular node in the parse tree, or text offset in the file. Edits are specified textually, however, since any edit may affect the structure of the parse tree, perhaps invalidating other actions' edits. Edits consist of add-text and delete-text commands and are processed in reverse order (back to front). By processing the edits in reverse order, the character positions designating the location of each edit do not need to be updated as the edits are applied to the document. Once the edits occur,

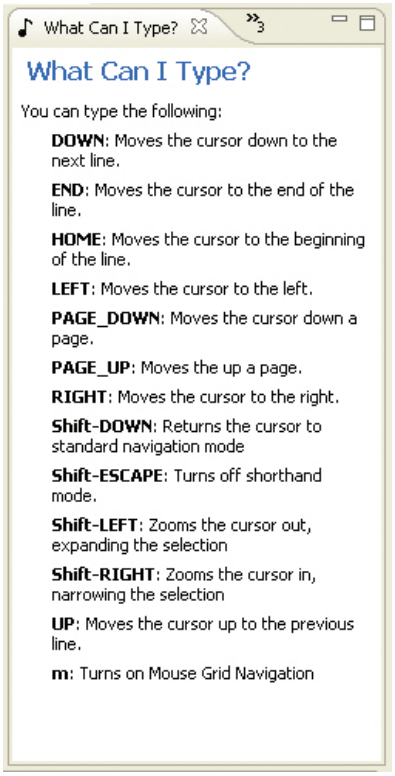


Figure 7.2: A screenshot of the What Can I Type? view. Keystrokes are listed on the left, and descriptions of the keystrokes' actions are on the right.

the program is reparsed, the box is repositioned and the cycle begins again.

Shorthand has over 50 actions available, and has not yet been finished. To ease the learning curve for all these actions, Shorthand provides a What Can I Type? view shown in Figure 7.2. This view shows all the keystrokes available for the current cursor location and describes what they do.

Shorthand was conceived by John J. Jordan. Jordan built both a prototype version and the Eclipse plug-in. The author collaborated with Jordan to extend the design and implementation to make it appropriate for programming by voice.

7.3.3 Speech Recognition Plug-in

Speech recognizers support two modes of interaction, command mode and dictation mode. Command mode requires the application to supply a set of finite-state grammars containing all possible commands in the system. When the microphone is turned on and the user speaks a phrase that

is recognized by the command grammar, the Java Speech API (JSAPI) invokes a callback with the recognized command. In dictation mode, however, all spoken words are allowed. As each word or group of words is recognized, another callback is invoked by JSAPI to alert the application. Command mode and dictation mode may be intermixed and both turned on simultaneously. The first words spoken after a pause are processed as both dictation and commands. If the words do not form a command, they are returned to the application as dictation. In order to switch from dictation to commands, the user must pause for a moment to reset the recognizer's command recognizer. Command mode tends to be more accurate due to the constricted vocabulary. Dictation mode allows all possible words to be uttered, restricted only by a hidden Markov model trained on documents that the user has supplied to the recognizer.

SPEED provides speech recognition control over Shorthand. Each action map in Shorthand is represented by a rule in a speech command grammar. Each action in Shorthand can be activated by a spoken phrase. The spoken phrases defined in the action map XML file are strung together into an string of alternative tokens. Each rule may be turned on and off independently, so as Shorthand moves its box around the program, the speech recognizer plug-in enables and disables rules to correspond to the active commands. Activated boxes in Shorthand allow for freeform dictation. When a box is activated, dictation recognition is turned on and recognized words are inserted into the box.

To improve recognition accuracy of dictation, a corpus of 60 Java files, converted from Java to Spoken Java using the Java to Spoken Java translator described in Section 3.3, was used to retrain the Dragon NaturallySpeaking speech recognizer. Each file was scanned for new vocabulary words and for pairs and triplets of words. This new ground truth data is used to alter the hidden Markov model so that when a user speaks some Spoken Java code, for example, "field Object object," the dictation engine is more likely to return the correct words than if it were trained simply on natural language documents.

7.3.4 Context-Sensitive Mouse Grid

As we reported in Section 2.2.1, navigation with voice recognition suffers from excessive cognitive load and delay. This causes an unsatisfactory user experience. One idea that came out of our study was the context-sensitive mouse grid.

The beauty of the commercial speech recognition tools' mouse grid command (see Chapter 2.2.1) is its simplicity and speed in quickly moving the cursor to a precise location on the screen.

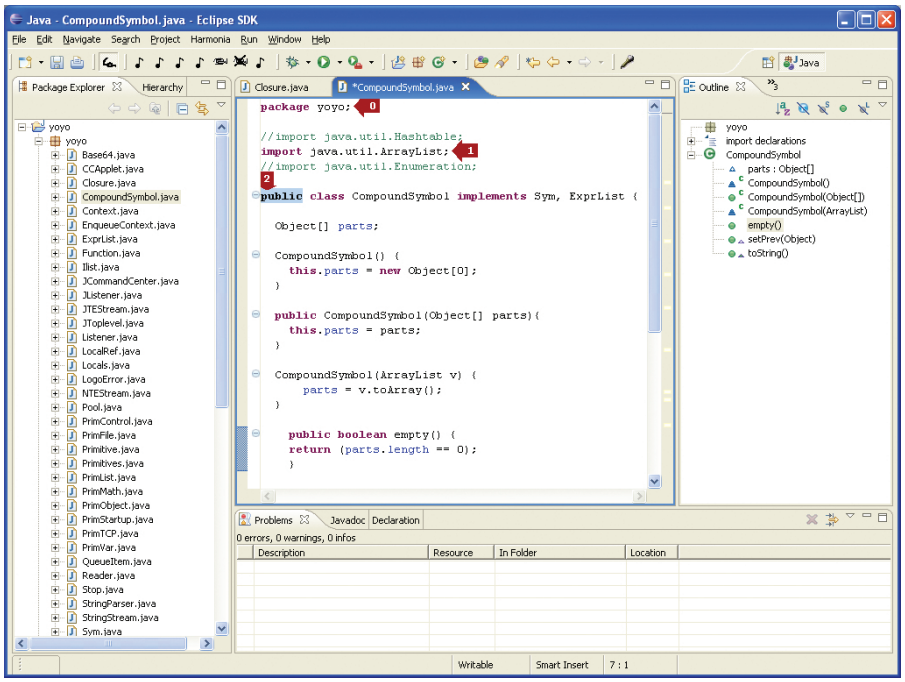
However, it does not "know" what is on the screen. In addition, people do not and cannot remember unique terms for all semantically important areas visible on the screen (even worse, icons usually have no text alternative).

If mouse grid could be context-sensitive and be able to tag semantically important areas of the screen, such as the menu bar, toolbar, scrollbar, document with textual names or numbers, then, when the user invoked mouse grid, visible tags would appear on the screen naming the area of the screen which can be zoomed in upon. The zoomed-in mouse grid would then tag more important areas within that initial area (such as each individual menu, or each section of the toolbar, or the large areas of the scrollbar (thumb, up arrow, down arrow, page up section, page down section) etc. If the user zoomed in on text, the text would be annotated with numbered tags to visualize the paragraph, then sentences/lines, then words (and characters if necessary) to enable the user to precisely drill down and move the cursor where he wants it to go.

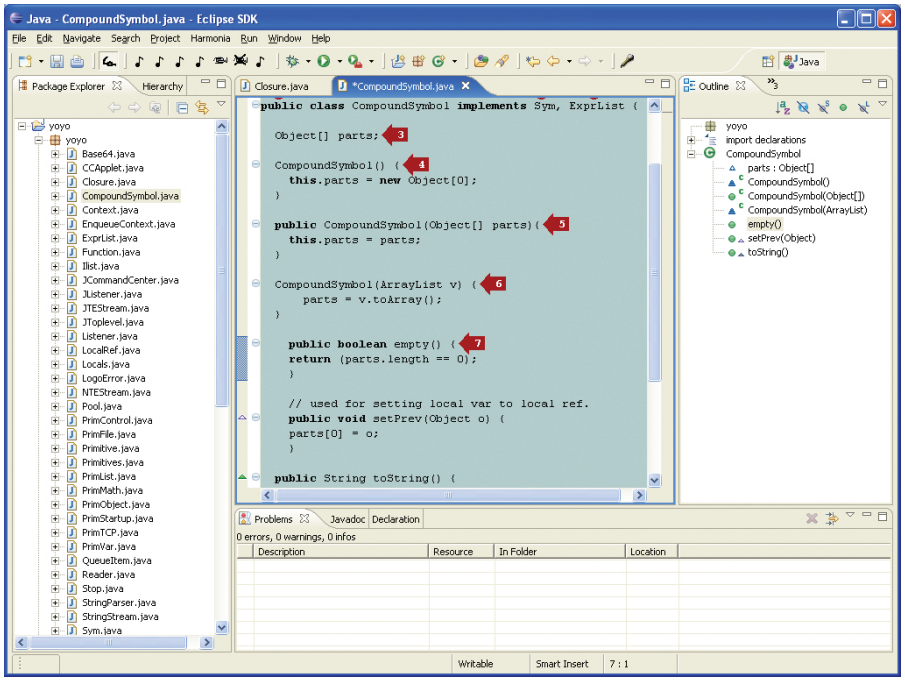
We have implemented this idea in the context of the SPEED program editor. The user can easily select by number (using keyboard or voice) any program construct shown on the screen.

Context-sensitive mouse grid is based on the structure of the program being edited. Top-level structures that are visible on screen are enumerated and labeled by red arrows with numbers inside. This can be seen in Figure 7.3. For example, the package line, each import and each top-level class visible on screen is labeled. The user can then type or say the number to zoom in on that program element. A new list of numbered arrows is generated for the substructure found within. If the desired element is not on-screen, the user can page up or page down to see more of the program. The user selects numbers until the correct program structure is highlighted, at which point he hits the Enter key or say "select." The Shorthand box is then moved to the selection, and an appropriate set of commands is enabled.

Java programs are not very deep hierarchically, so many structures can be selected with very few numbers. Cognitive load is decreased because program elements need only to be read, not spoken, to be selected. Delay in recognition is also improved since the user can point at the absolute location of the program element without having to repeatedly speak the same command. One caveat of context-sensitive mouse grid is that the numbers shown on screen, while deterministic, are not predictable. Since they depend on the program, as the program changes, the numbers change. Since they only apply to the part of the program that is visible on screen, as the user scrolls, the set of shown numbers will change as well. Traditional mouse grid would not be as useful as one might think, since many program structures exist at the same point on the screen. Selecting the point would necessarily need to be followed by expand selection Shorthand commands to navigate hierarchically



(a)



(b)

Figure 7.3: Part (a) shows Context-Sensitive Mouse Grid just after being invoked. Three arrows label the top-level structures in the file. Part (b) shows the same file after the number 2 (the class definition) has been picked. The entire class has been highlighted, and new arrows appear on the structural elements of the class that can be chosen next.

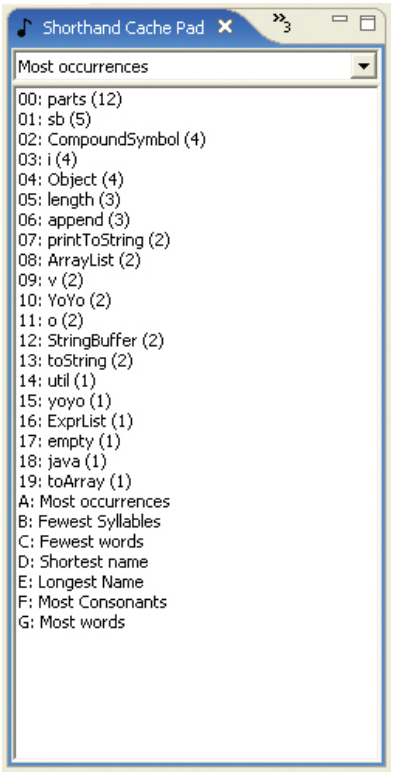


Figure 7.4: A screenshot of the Cache Pad. It shows the twenty most common words in the currently edited Java file.

to the proper structure. Context-sensitive mouse grid conflates the two operations resulting in fewer commands spoken.

7.3.5 Cache Pad

Speech recognizers are notorious for mangling the recognition of uncommon words, especially those used in new contexts that do not conform to the models of documents that have been used for recognizer training. In addition, since the programmer can not easily verbalize spelling, capitalization or the spaces between words, considerable effort can be expended to wrangle an identifier name into the proper form. After going through this process, the programmer certainly does not want to do it again. The Cache Pad is a numbered view of twenty recently spoken identifier names (found through a search in the parse tree) that may be inserted into the program by referring to the word's number. The CachePad, shown in Figure 7.4 is activated during any activated box, which includes all identifiers in the languages. Number recognition is much more accurate

and consistent than recognition of words, lowering considerably the effort requiring to reuse an identifier.

Since the Cache Pad holds only a limited number of words, the programmer can choose, at run-time, from a number of Cache Pad policies that specify how to fill the twenty slots. The simplest approach is to take the twenty most recently uttered identifiers in the hope that their use enjoys temporal locality. However, perhaps the hardest words to say should be in the Cache Pad, for instance, the longest words, the ones with the most number of subwords or the most common words. Or perhaps the words immediately surrounding the cursor, taking advantage of spatial locality. Cache Pad could tie into the code completion facility in Eclipse, and try to predict the most frequently used identifiers that come next after the cursor.

Learning what the best policy is would be easy if we could study logs of programmers working on their programs. At any point in time, the best policy would predict the next twenty words the programmer will say. Until this study is undertaken, we enable the programmer to choose the policy he would like by inserting the policies into a second panel in the Cache Pad and labeling them with letters. We can look at how programmers choose which Cache Pad policy works best for them and perhaps find a good set of choices to use for defaults.

7.3.6 What Can I Say?

An important result from Christian, Kules, Shneiderman and Youssef showed that verbalizing commands takes up more short-term memory than typing the same commands [15]. This is caused by cognitive interference between speaking and short-term memory. As speaking is the primary form of input in SPEED, there may be two problems for programmers: 1) They could forget what they are allowed to say. 2) They could forget their program as they are writing it. To address the first issue, we created a What Can I Say? analog to the What Can I Type? view which displays all of the legal commands that one can say at any given time. This view, shown in Figure 7.5, updates as the Shorthand box is moved around. The second issue is addressed through the interaction mode for SPEED. While creating and editing programs, developers receive visual feedback of what they are saying. They edit a program they can see on the screen, so they are always aware of the context of what they are saying. The absence of this context helped to cause the problems reported by Christian et. al. above, since the short-term memory of the context is what was interfering with the speech.

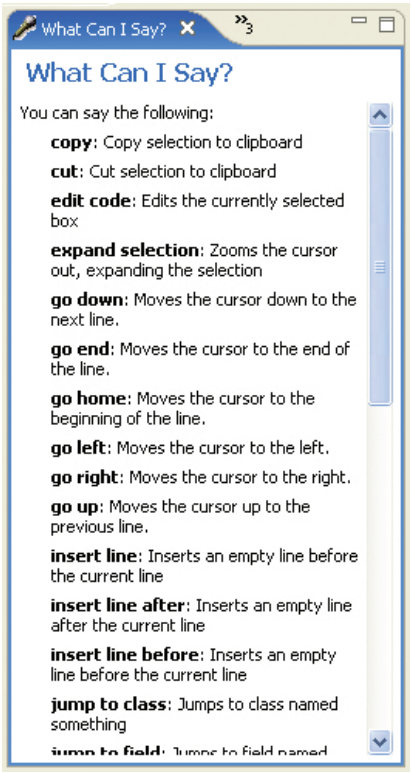


Figure 7.5: A screenshot of the What Can I Say? view. Spoken phrases are listed on the left, and descriptions of the phrases' actions are on the right.

7.3.7 How Do I Say That?

Learning to speak a new natural language, especially as an adult, can be a trying experience. In addition to learning new vocabulary, people must also learn new grammatical structures, culture-specific idioms and a large body of literature in order to become truly fluent. Similar problems plague software engineers who learn several programming languages during their computer science education. In addition to learning the vocabulary and grammar for a particular programming language, they must also learn the "culture" of logic and algorithms and decipher many existing programs and APIs to learn how they work and how they may be used for their own purposes.

Learning to speak programs out loud will be easier for those who already know how to write code. Since programmers already know the written language, libraries and code examples, all that is left to learn is new vocabulary and grammar. However, training will be required to teach programmers how to use this new input modality. In addition to training by a human teacher, we have provided the How Do I Say That? feature to SPEED. By moving the Shorthand box onto a

language construct in the editor, the programmer can ask for a dialog box to show the construct's translation to Spoken Java. This Spoken Java text can be uttered exactly as written on screen to enter the same code. By using How Do I Say That? on a variety of program constructs, programmers can grow comfortable with speaking code out loud, as well as have a crutch to fall back on when they do not remember the right words or phrase structure to use.

We had hoped to incorporate a Java to Spoken Java text-to-speech system to speak programs out loud, but we did not have the time to work on it. This spoken feedback would provide an alternate means of teaching and reinforcing proper Spoken Java vocabulary and grammar. A simple approach would translate Java code to Spoken Java, which removes most punctuation, but preserves all of the natural language words, and pass the translated code to a text-to-speech engine that comes with most commercial speech recognizers. To make this Spoken Java speech synthesis more useful in practice, we would adopt approaches taken by Smith and Francioni in their work with blind programmers [91, 32].

7.3.8 Phonetic Identifier-Based Search

Phonetic identifier-based search technique is similar to a Google search for words in the program, without regard for correct spelling, capitalization, or word boundaries. All of these relaxations of a strict match are important because users cannot say them easily in a voice-based search tool. This tool uses phonetic (Soundex) searching rather than exact character matches. The order of the words does not matter, making the search more similar to a Google search, rather than a traditional word processor search. In addition, this find tool is non-modal, to eliminate extraneous dialog box interactions (bring up the dialog box, click OK to dismiss, click Find Next to continue the search) which slow down voice control. When invoked, the search tool brings up all results of the search and displays them in summarized form (with a sentence of context above and below) on one side of the screen. Each match is numbered and the user speaks the number of the desired match to go to the location.

We have not yet implemented this search feature in SPEED.

7.4 Spoken Java Command Language

SPEED employs a command language for navigating around the program and editing code. There are several categories of commands listed in Tables 7.1 and 7.2. The first category is

Navigation Between Boxes		
Go Up	Jump to Class *class name*	Expand Selection
Go Down	Jump to Method *method name*	Shrink Selection
Go Left	Jump to Field *field name*	Mouse Grid
Go Right		

Editing a Box
Edit Code
Insert Here
Insert After
Insert Before
Delete
Insert Line

Editing Within a Box		
Go Left	Delete Left	Done
Go Left Word	Delete Left Word	Cancel
Go Right	Delete Right	Go Home
Go Right Word	Delete Right Word	Go End
Select All	Insert Line	Translate That
Select Left Word	Space	Cap That
Select Right Word	Scratch That	Lowercase That

Table 7.1: Programming by Voice Commands Part 1

Navigation between Boxes. This category contains commands to move the Shorthand box around the program. Jump To commands enable direct navigation to a named entity in the program. Expand and Shrink selection change the size of the currently selected expression. These selections move up and down the parse tree, skipping chain productions (productions with only one symbol in their right-hand-side). Mouse Grid activates the context-sensitive mouse grid feature.

Box Editing Commands contains the commands for modally editing or creating code with Spoken Java. Edit Code edits existing Java code, while the Insert commands create new Java code. Delete gets rid of the currently selected code, and Insert Line inserts a carriage return before the currently selected box.

Once a box is used to edit or create Spoken Java code, commands in the **Editing Within a Box** category are available to manipulate the cursor. The Go Left and Go Right commands go left and right one character. The Go Left Word and Go Right Word commands go a full word. Go Home and Go End move the full length of the box. Delete commands are analogous. When the user makes a mistake, he can say Scratch That to undo the last utterance, or say Select All, Select Left Word or Select Right Word to highlight some text and speak it over again. Since carriage returns and spaces are difficult to say, commands to insert these are available. Cap That and Lowercase That commands change the capitalization of the first letter of a word, useful for camel-cased identifier names. When the user is done talking, he says Done to accept the changes, and Cancel to revert back to the original. After he accepts the changes, the user says Translate That to pop up a list of

Code Template Insertion
Insert Class
Insert Interface
Insert Method
Insert Constructor
Insert Field
Insert Comment
Add Initializer

Miscellaneous Commands	
Cut This	Cap That
Copy This	Lowercase That
Paste This	
Undo	Redo

Table 7.2: Programming by Voice Commands Part 2

Java interpretations of the Spoken Java input.

Code Template commands are used to insert boiler plate code, textually, into the code buffer. New entities are created with default names and pretty printing. Add Initializer adds a default initialization routine to variable declarations.

Finally, there are **Miscellaneous Commands** to operate the clipboard, and undo/redo functionality. Cap That and Lowercase that can change the capitalization of identifiers.

Chapter 8

SPEED Evaluation

SPEED is a novel programming environment, both for its spoken user interface and for the Spoken Java language. While the design of the programming environment was based on an initial set of user studies, it is important to conduct follow-up studies to validate the design and discover problems that can be addressed in future work. We consider our work successful if programmers can use SPEED to write and edit code by voice in a reasonable amount of time. In this chapter, we describe a study we conducted to have expert Java programmers write and edit code in SPEED.

In our study, we asked programmers to build a data structure and associated algorithms. This study helped us to understand how developers mix Spoken Java with the Spoken Java command language as well as showed us the kinds of commands the programmers use to manipulate the code in the editor. In addition, we were able to classify the mistakes that programmers and our system made that affect non-contiguous code entry and edits. By coding a simple data structure, any observed difficulties were likely to be an indicator of the programmers' efforts to learn and use SPEED rather than of their efforts to create the program itself.

Our user study was conducted in two distinct sessions. The sessions were identical except for voice recognition technology. In the first, we used Dragon NaturallySpeaking. In the second, we used a non-programmer human to listen to the programmer from behind a curtain and type in what the programmer said. The first session represents what can be done with the state-of-the-art in voice recognition tools with minimal training. However, since the goal of our study was to learn how programmers used SPEED, and not to explore the limitations of voice recognition technology, our use of a human speech recognizer in the second session illustrates how SPEED could be used when the voice recognition accuracy is as close to perfect as it can get.

8.1 Participants

The participants in the study were expert Java programmers with an average of twelve years of experience, drawn from the software development industry. Most of the programmers had never used speech recognition software before; those who had used it played with it for a short time, but quickly abandoned it because of poor accuracy. None of the participants had motor disabilities which would make typing difficult. The first session had three people; the second session had two.

Participants had been using Java-based IDEs for around a year; most used Eclipse. All of the participants used these IDEs to write code, edit code, and browse through code. Some also used them for building applications and running tests. In their everyday work, participants did not spend an equal amount of time coding versus reading. The split varied by the phase of the project or the programmer's own job in the project team. At times, all the programmer's time was spent writing new code, sometimes it was 100% editing code. Navigation and browsing usually took around 30-50% of a programmer's time no matter what their primary task was.

When asked whether they would consider using voice recognition for programming, most were apprehensive, pointing out problems with noise pollution (many work in cubicle farms, which are very noisy to begin with), cognitive interference (speaking interferes with thinking), and a potential for wasting the use of their hands while speaking. One person would only use voice recognition if everyone else were to do it. One person might use it at home if he worked alone, where he could not bother anyone else.

8.2 User Tasks

Developers were asked to perform a programming task in 20 minutes: create a linked list data structure with an append method. Linked lists usually contain two fields, one pointing at the element, and another pointing at the rest of the list. The constructor is used to build a linked list node. The append function takes two linked list parameters and merges them. An alternate form of the linked list has a nested node class to hold the list element and next pointer. This list also contains a pointer to the end of the list so that extending the list can be done in constant time.

8.3 Experimental Setup

SPEED was deployed on a Pentium 4 3.2 GHz computer with 2 GB of RAM. Programmers used a Plantronics DSP-300 microphone. Screen captures of the experimental sessions were taken with Camtasia Studio 3. The developers' voices were recorded using Quicktime Pro on an Apple PowerBook computer with the microphone from an iSight video camera.

The first session's software developers went through a 15-minute Dragon NaturallySpeaking 8 voice training process and a 15-minute SPEED training session before they began their tasks. To augment the What Can I Say? feature of SPEED, developers were given a paper crib sheet with commonly used commands printed in a big font (this was in addition to the What Can I Say? view in SPEED itself). The second session skipped the 15-minute voice recognizer training process.

During the second session, a human volunteer was recruited to *be* a voice recognizer. He was set up behind the study participant, facing the opposite direction. His display was connected to the study computer via VNC [81], allowing both him and the study participant to see what was happening on the screen at the same time. For performance purposes, instead of using VNC to control the mouse and keyboard, the human voice recognizer used a second keyboard and mouse connected to the study computer via USB cable.

The human voice recognizer was trained for 45 minutes prior to the studies to practice interpreting Spoken Java commands, and when to just type in what the user spoke. To assist in translation, the human was also given a crib sheet indicating the mapping between command phrases and keystrokes used to activate those commands. Punctuation and other non-words were entered in an arbitrary way by the human as he saw fit. Depending on the brand of voice recognizer, a software client may receive a punctuation mark or the spelled out punctuation. So, this setup is similar to the kind of text that the voice recognizers return.

8.4 Evaluation Metrics

The users' programming sessions were analyzed on a number of metrics:

- Number of Commands or Dictated Phrases
- Number of Correctly Interpreted Recognition Events
- Number of Commands Uttered

- Number of Dictation Words Uttered
- Features Used (further subdivided into code template commands, dictation of identifiers or statements, navigation commands, editing commands, and commands used to fix mistakes (caused by any reason).
- Number of Speech Recognizer Mistakes (further subdivided into mistakes where extra words were inserted vs. incorrect words inserted vs. when the recognizer did not respond to what was said)
- Number and Types of SPEED Mistakes (further subdivided into mistakes where the system should have interpreted the utterance properly but did not due to a bug, the number of mistakes due to design flaws in the system, and the number of times that SPEED crashed)
- Number of User Mistakes (further subdivided into mistakes where user did not know what to say, user used the wrong command, and user said the right command but said it ungrammatically or used the wrong words)

Metrics were coded by one viewer watching the screen capture while listening to the audio recording of participants performing the tasks. Each participant's video and audio recording was played back three times to confirm the measurements.

8.5 Hypotheses

SPEED is designed to be a voice-enabled structure editor with modal activation of Spoken Java dictation. We hypothesize that SPEED users will follow a typical programming pattern: they will navigate through the document to a desired insertion point, activate Spoken Java, and add new code or edit existing code. A few of the editing commands will be used in the majority of situations, so we anticipate that learning the commands should take only a few repetitions. Based on our earlier user studies, we anticipate that speaking the Spoken Java language should be intuitive and natural for programmers without any training. Our GOMS analysis predicts that programmers should work more slowly when programming by voice than by keyboard due to the slow speed of speech recognition compared to typing. Finally, based on the results of our document navigation study, we think that programmers will sometimes make the kinds of mistakes where they do not remember what they were doing because of anticipated cognitive interference between speaking and thinking about code.

8.6 Results and Discussion

We present the data recorded from the study according to the metrics described above.

		User 1	User 2	User 3	User 4	User 5
		Machine Voice Recognizer			Human Voice Recognizer	
Commands or Dictated Phrases		131	125	275	102	105
Correctly Recognized Speech		66	97	164	81	92
Commands		73	105	206	68	70
Dictated Phrases		58	20	69	34	35
Features Used	**Code Template**	4	10	7	7	4
	Dictation	10	14	16	14	9
	Navigation	16	43	71	25	29
	Editing	49	51	130	36	36
	Fix Mistakes	18	4	60	1	1
VR Mistakes	**Extra Words**	6	0	3	0	0
	Incorrect Words	8	3	34	1	3
	Did Not Hear	13	15	42	1	0
SPEED Mistakes	**Bug**	5	0	2	2	4
	Design Flaw	1	2	0	2	0
	Crash	4	6	5	2	4
User Mistakes	**Did Not Know**	2	0	2	5	2
	Wrong	2	2	3	0	3
	Ungrammatical	2	2	11	1	0

Table 8.1: Data recorded from SPEED User Study from all participants.

8.6.1 Speed and Accuracy

The first session of programmers had a decidedly different experience than the second session. They were stymied by the speed and accuracy of the voice recognition software and were only able to accomplish a portion of the code creation task in the 30 minutes allotted. They could create a class, some fields and a constructor, but were not able to fill in the code for the constructor or create any methods. The second group, which used the human voice recognizer, enjoyed almost error-free recognition, and were able to complete far more of the code creation task. They were able to complete two classes (a list node and a list class), each with several fields, a constructor to initialize the fields and the beginnings of an append method.

The accuracy of the machine voice recognizer was abysmal. At times, the recognizer would just not hear anything the programmers said. Other times, it would recognize a series of commands perfectly. Dictation, however, was fairly unusable. Breathing was often interpreted as a single syllable word, so programmers learned to use the mute button on the microphone to

prevent this from happening. Often, words were recognized poorly, coming close to what the user wanted, but not close enough. For example, “linked” came out as “links.” Text editing commands (as opposed to program editing commands) provided by SPEED were minimal, though similar to those provided by Dragon NaturallySpeaking, and were tedious to use, so most programmers used “scratch that” to undo the utterance and try it again.

For two programmers, we trained the speech recognizer on a corpus of Spoken Java programs. However, out of several hundred thousand words, this corpus contained a few, very common words with uncommon spellings (containing punctuation in strange places, odd capitalization), which caused the speech recognizer to generate them over and over again. In addition, many identifiers in the corpus were unique to that corpus and not applicable for the linked list program. Most of the study participants did not have their recognizers trained on this corpus.

The second group reported almost error-free recognition due to the human speech recognizer. This contributed significantly to a reduction in the number of commands uttered to fix mistakes. Recognition delay was about the same for both groups, around 0.5 to 0.75 seconds. The reaction time of our human recognizer was about equal to the processing time of the computer. We imagine that faster computers will reduce this processing time further, while the human reaction time will remain the same.

A major component of the software speed problem was Camtasia Studio. When it was recording, the software slowed down by at least three times. One participant who got to use SPEED for a short time without screen capture reported that the speed was perfectly fine.

Due to the speed and accuracy problems, participants in the first session adopted a stop-and-go pattern of speaking and waiting for the results. Their mistrust of the voice recognizer caused them to program very slowly and increased their frustration with SPEED. The second group were able to speak at a normal pace, and often paused in the middle of commands as they were uttered, something a machine speech recognizer would have never recognized correctly.

8.6.2 Spoken Java Commands

All users learned the SPEED commands fairly quickly, requiring only one or two repetitions of each command to be able to use it without looking at their crib sheet. Command error rates caused by user error were fairly low for all participants. Out of around 148 commands each, the average number of errors was around 6 (one user made 26 errors), giving an average user error rate of 5.5%.

	User 1	User 2	User 3	User 4	User 5
	Machine Voice Recognizer			Human Voice Recognizer	
% Correct Recognition Events	50.4	77.6	59.6	79.4	87.6
% Commands To Fix Errors	13.7	3.2	21.8	1.0	1.0
% Commands For Navigation	12.2	34.4	25.8	24.5	27.6
% Commands For Editing	37.4	40.8	47.3	35.2	34.3
% Commands Inserting Code Template	3.1	8.0	2.5	6.9	3.8
% Commands Starting a Dictation	7.6	11.2	5.8	13.7	8.6

Table 8.2: Distribution of Spoken Java commands spoken for various purposes.

In Table 8.2, we see a breakdown of the distribution of Spoken Java commands used for different purposes. The first few users had recognition error rates between 23% and 50%, which made the system unusable. With the human recognizer, the final two participants enjoyed less than 21% error rate. The errors made by these users were their own, however; they were not caused by misrecognition, which because it was under the users' control, was much easier to accept.

The two most used command types were code template insertion and edits of field, method and type names. Code template insertion was seen as providing a lot of text for very few words. Editing was a simple repetitive process that involved navigating to the word, saying "edit this," saying the new name, and then "done." Most new names were single words, enabling SPEED to automatically translate what users said from Spoken Java to Java without any need for interaction.

The majority of commands spoken were for editing. Editing commands were mainly used to take the place of string editing facilities that are usually performed by repetitive single keypresses, such as deleting a character from a word, inserting spaces in between words, or changing the capitalization of a word. As predicted by our GOMS analysis in Section 2.3.2, spoken commands that result in few letters changing on the screen cannot compete with typing for efficiency. We have no good ideas for reducing the complexity of these actions other than to use the keyboard.

Spoken commands need to have a big payoff in modifying the program to be efficient. Thus, the small percentage of code template commands achieved the largest payoff for the programmer, and are almost directly correlated with the number of program structures created by the study participants. Code dictation was almost always used to edit the name of a field, method or type name. Since most new names were single words, SPEED was able to automatically translate what users said from Spoken Java to Java without any need for interaction. Multi-word identifiers were concatenated automatically, using our speech-away programming language analyses, enabling developers to skip some tedious editing commands to affect spacing and capitalization. Capitaliza-

tion of the first letter of a word (encoding a style convention in Java that class names start with a capital letter) was not performed automatically, and thus some of the remaining editing commands were required to change the initial letter of an identifier to capitalize it.

Two users used context-sensitive mouse grid for navigation, and both felt that this was a huge improvement over the simple arrow-key-like navigation commands otherwise provided.

Participants had suggestions for better commands. One wanted a Jump To command where the user could read code off the screen and have the SPEED cursor jump there. This is very similar to Select and Say, from Dragon NaturallySpeaking. Several wished that the code templates could take a parameter with the name of the item being created. For example, "insert field element" instead of "insert field." Code templates only worked on blank lines. One user wanted the commands to work anywhere, since for example, there is only one logical place to insert a field when you are inside a class, no matter what you are currently selecting in that class.

A few asked to customize both the command names to insert code as well as what code templates were available. One participant explicitly said there should be no customization in order to make it easier for all users of a speech programming tool to learn the same language, as well as to make it easy to move from one speech-based programming environment to the next.

8.6.3 Speaking Code

Participants were apprehensive about speaking the natural language words in the program when dictating code, but not when saying identifier names. In fact, most dictation utterances were for identifiers. One called the idea of dictating code "strange." Four preferred to describe the code instead of dictating it. This is an unexpected, but interesting finding. One felt that describing the code was a higher-level of programming, apparently concluding that if higher-level programming is a good idea, then describing code must be as well.

Two participants felt that tandem use of keyboard and voice would potentially be more efficient than either alone. Without specialized navigation commands, voice is inefficient at moving the cursor to a particular location on screen, especially inside pure-text regions. But a keyboard and mouse make this simple. For code entry, voice could be more economical through code templates. Code templates are available by keyboard in Eclipse, but participants had trouble remembering all the proper keywords to activate them. They thought the voice commands were much easier to remember.

One wished for integration of voice with Eclipse' code completion feature. When the user

was entering a method parameter's type, a list of all types could be shown in a popup menu. As the user spoke the words composing the type name, the menu would be filtered to only show matches. Once there was only one left, the user could then select it.

8.6.4 Evaluation by Participants

The participants concluded that none of them would use this programming environment for daily coding, especially with the poor accuracy provided by the machine voice recognizer. However, they all would consider using the software if they came down with RSI, worked from home, or were stuck in a hands-free environment, such as while pacing around the room, or sitting with their feet up on the desk with the keyboard far away. In spite of their reluctance to use this software, all programmers noted that since coding was not a significant component of their daily work, using a voice-based programming environment would not have a significant effect on their efficiency as a programmer.

8.7 Future SPEED Designs

The next step in a user-centered approach for programming-by-voice is to learn from our study and iterate the design. Based on our study, there are several features of SPEED that could be improved. Code template commands were found to be easy to learn, so we would increase the number of templates to cover program statements, as well as improve the customizability of the templates to let users build their own. Editing commands for string edits have proven to be impractical to conduct by voice. We can look into enhancing our program analyses to better predict likely spelling, capitalization and word spacing for names and identifiers commonly used by programmers in their own programs. In addition, we could add a machine learning algorithm to analyze the names used in a particular developer's programs and use that information to help predict the most likely spellings for spoken identifiers.

Participants' reluctance to use code dictation services needs to be further explored. We need to understand the source of the discomfort; perhaps it could be overcome by further training. While code dictation would not be more efficient than code templates, it would be useful for code constructs that do not have programmer-understood names and for complex program constructs that programmers will only use once. Finally, the entire IDE will need to be speech-enabled. We can build off the SpeechClipse work to achieve this goal [90].

8.8 User Study Improvements

The user study showed that voice recognition performance after 15 minutes of training is too poor for use in a study. Programmers would have to train for many hours before recognition accuracy would improve enough to be usable. Obviously, continuing to use a human voice recognizer distorts the accuracy results by assuming a perfect recognizer. One approach to fixing this could be to use speaker-independent voice recognition software. These are designed to work without any training, although most of them do not support free dictation, only commands.

Our study looked primarily at writing new code. Editing commands were used to fix mistakes and edit identifier names, but clearly, large-scale code maintenance, and even small code motion operations would need to be speech-enabled and evaluated. Operations that invoke IDE commands have more traditional implementations (all recognizers can speech-enable menu items and GUI buttons), but the more common operations could benefit from evaluation through a user study.

Few users tried code navigation via context-sensitive mouse grid. For the first three participants, recognition of numbers was not working at all, preventing its use. The next two users used it once or twice (more often in the training session) and would use it again. We would like to evaluate this feature more completely, out of the context of code manipulation, to see how understandable its code selections are, and to figure out whether it enables navigation to all desirable program points. This will require another user study.

8.9 Summary

The five programmers each learned to program by voice using SPEED in their study session. The commands were each learned with one or two uses. The inaccuracy of the voice recognition adversely affected programmers' performance, but did not change how they used the SPEED commands. Programmers noticed that they were less efficient with voice-based programming, independent of recognition errors, though this inefficiency would not adversely affect their job performance. This was predicted by our GOMS analysis (see Section 2.3.2) and confirmed by the programmers in their post-study interviews. Several classes of improvements can be made to streamline the commands. Some of these involve combining separate commands that are always used together into a single command. Others include voice-enabling the rest of the IDE outside of the program editor to enable use-def program navigation and Eclipse-provided program refac-

torings. Assuming the social problems associated with voice-based programming in a work environment can be solved, programming by voice does appear to be a viable alternative to typing for creating, editing and navigating programs.

Chapter 9

Commenting By Voice

In this chapter, we discuss commenting by voice, the technique of documenting program code using voice recognition. We explore possible reasons for the lack of documentation in existing software, and propose commenting by voice as a possible solution. By employing two input modalities: keyboard and speech, it may be possible to overcome physical interference between coding and commenting activities. Then we explore several commenting by voice scenarios to explain its utility in real-life situations. We end by discussing a prototype implementation of commenting by voice and propose an experimental study to validate the idea.

9.1 Documentation of Software

Writing documentation is a perpetual exercise for the creators of software artifacts. For end-users, documentation is a key resource to learn how to use the artifact, but for the developer, documentation enables much more – not only the ability to understand someone else's code, but to document one's own thought processes regarding the architecture of the artifact and the code itself. Unfortunately, programmers usually do not create enough documentation, nor high enough quality documentation to replace an in-person discussion of the code. We believe this to be caused not by the inherent "laziness" of programmers to document their code, but by physical interference in the commenting activity, since both programming and commenting utilize the same input channel: the keyboard (or in voice-based programming environments, voice recognizers). Programmers should have a tool that enables them to create code comments using audio recording and speech recognition at the same time as they are typing in their program code by keyboard. These audio comments can then be inserted in their code with the same status as textual comments. We hope that by paralleliz-

ing their input channels, programmers will comment their code more, and in doing so, enable others to better understand the original thought processes involved in the coding task. Programmers are quite expressive writing code; if given a chance, we hope that they become as expressive talking about it.

Documentation about software artifacts is at least as ubiquitous as the program code required to implement them. Programmers need many forms of documentation to do their work. For instance, programmers use API documentation to understand how to use libraries created on-site and purchased from outside vendors. Application designers create system architecture documentation to describe how all the components of a system fit together. Developers create engineering specifications to refine the client specifications in order to explicitly state the myriad implicit details that were left out.

In the source code itself, programmers create header documentation to describe the purpose of a file. Comments about code are also ubiquitous, and can span the gamut from a high-level description of an algorithm to low-level explanations for non-intuitive (read: clever) code. In this section, we will concentrate on code documentation in the form of program comments. We will justify this focus later in this section.

If documentation is so pervasive, why then is it almost universally perceived as being of such low quality? Communication and writing skills have often been found lacking in new college graduates in computer science. Since programmers have such bad writing skills, McArthur claims that they should not be the ones writing any documentation for end-users of a system [66].

End-users are not the only consumers of documentation, as we saw above. Programmers themselves use documentation for many purposes, so we should ask, do programmers write good documentation for their own kind, and if so, what benefits are derived from the activity and the product? Detienne's studies [21] indicate that programmers who write comments before they begin coding seem to perform better on code comprehension tasks. Comments appear to aid in chunking, the process of grouping pieces of knowledge together. Unfortunately, there is a pitfall to commenting that can affect comprehension: comment management. When the comments do not appear in the code, but in a separate document, one runs the risk, especially in larger software projects, of the documentation getting out of sync with the code that it is describing [59].

Sometimes there is too much documentation that focuses on details that experts find useless. Often this sort of documentation is written by novices because they themselves do not understand how a program works, so when told to comment their code, they concentrate on the lexical and syntactic pieces that they do understand [13]. Why are these kinds of comments (about a pro-

gram's lexical and syntactic properties) less useful than comments about the semantics? Riecken, Koenemann-Belliveau and Roberston performed a study on expert programmers' intentions for their comments [85]. First, they found that experts communicated semantic rather than syntactic knowledge about their program in the comments. This was because syntactic knowledge was assumed to be already understood by any reader (even complicated syntax was perceived to be a rite of passage for novices to understand), whereas the semantic knowledge was the hard part to understand and therefore the most critical to convey.

The flip-side of the problem of too much documentation is too little. The fact that programmers do not write documentation until after they are done coding or perhaps never at all is well-known in the technology industry and is perhaps one instigating factor of the many new programming methodologies that pop up every few years to encourage programmers to comment more [28].

9.1.1 Solutions

We can solve some of these documentation problems rather easily. If programmers are bad writers, we can just hire technical writers to write the end-user documentation for them. Better programmer education as well is advocated to improve not only programmers' communications skills, but to improve their programming methodology and habits [111].

We can solve the out-of-sync documentation problem by inlining structured documentation in the code and processing it with a separate tool to generate the final documentation. Such a technique is used by JavaDoc [56] for the Java programming language.

Some feel that tool support would help programmers document their code more easily [12]. There is a long history of automatic commenting tools [26, 96] which derive a description of the code through program analysis. More recently, an interactive commenting tool for Prolog was developed [86] that enables the programmer to comment on each step of execution of a Prolog query and insert those comments back into the code. Work with commenting agents has also been reported [27] to help users design user interfaces.

9.1.2 Programmers Are Just Lazy

However comprehensive this research seems, there is still one zebra left in our herd of horses. Programmers are perceived to be *lazy*. The unspoken argument is that they do not document because they do not want to expend the effort. We argue that this strawman argument is correct,

but for the wrong reasons. Programmers are *not* lazy. In fact, programmers are anything but lazy. Who else would commit to the same long hours in pursuit of a bug or finishing off a feature before a deadline?

9.1.3 Or Are They?

We assert that programmers comment poorly because there is no good time to do it. There are only three times during coding when a programmer could comment her code: before she starts, while she is coding, and after she is finished. The kind of comments that can be written before the code are only blackbox comments – they must necessarily be descriptive of the intent and semantics of the code, for the lexical and syntactic structures have not yet been written. After the programmer has spent a few hours poring over her solution, and the code is completely written, she feels as if she knows the code like the back of her hands. It may even seem that there is no reason to comment code that appears so intuitively obvious (however non-obvious the code will appear in a week). Of course, the explanation is that "the source code *is* the documentation."

If the programmer can not be fully descriptive with her comments before she starts, and does not want to or forgot to comment her code after she finishes, that leaves only one time to comment: while she is coding. We argue that programmers do not comment while coding as often as they should because coding and commenting use the *same input channel*: the keyboard. Thus, in order to comment their code, they must necessarily stop coding, and vice versa. Even if they would like to be very descriptive with their comments about the actual code, in the end, it is the code that they get paid to write, not the comments.

9.1.4 Voice Comments

In this work, we aim to solve this input channel conflict. We enable the programmer to construct *voice comments* as they program by recording what the programmer says out loud into a headset microphone. We use a speech recognizer (IBM ViaVoice [39]) to translate the audio into text, and insert this audio/text combination into the code as a comment. Voice comments can be played back aurally or read visually at any time. The comments are saved (structurally) with the program document and restored when the document is reloaded.

By using the voice channel in addition to the keyboard channel, a programmer can talk about his code at the same time as he codes it by hand. It is similar to a Think Aloud study, in which participants are encouraged to talk about what they are doing while performing an action.

This enables an experimenter to gain insight into the thought processes involved in a task without cognitively interfering with the task itself. We speculate that utilizing both voice and keyboard input channels will enable a programmer to annotate her code with spoken utterances about the thought processes that are going on in her head as she designs and writes down the program. In the rest of this section, we present the design and implementation of the voice commenting project, as well as possible scenarios for its use in the real world. We then discuss experiments we would like to perform with novice and expert programmers to both better design the system and elucidate its impact on the programming process. Finally, we describe future work and conclude.

9.2 Scenarios

In this section, we present some possible scenarios of interaction with a prototype Voice Commenting tool. The first is drawn from an educational perspective, and the second from a code review perspective.

The notation that we use for visualizing the code is as follows. Program code is marked in `Courier font`. Traditional program commands are marked in *italics* and bordered by language-defined boundary tokens (e.g. Java uses */* A comment */*). Voice comments are marked in *italics* with $\ll$ and $\gg$ boundary tokens.

9.2.1 Education

It has long been the case that everyone grading a programming assignment laments that they do not understand how a student could have possibly come up with the answers that he did. Students do not comment their code enough, and the code itself is usually written in a language or style that is particularly obtuse. Also, the submitted copy represents only a tiny fraction of the code that the student actually typed in while trying to make his project.

Would it not be nice if you could follow a student's thought process along from beginning to end, and see not just the end product of his efforts, but all intermediate stages in between? Even better, if you could not only watch his code develop, but you could also know what he was thinking when he wrote it? That would give a teacher/grader much more information to work with in order to understand how a student developed his code, and more easily identify where he went wrong (and, where he had a great flash of insight!).

Consider a second-year computer science student working on a Java programming assign-

ment to implement a linked list data structure. The student must define the appropriate Java class, but first thinks out loud about what he needs to do.

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

Then the student types in the class declaration:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
}
```

The student next states out loud that he knows one of the fields to add to the class:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
}
```

And, he then defines the value:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
}
```

At this point, the student wavers a bit. He is not sure how to complete the data structure.

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
```

≪And I know there's another to continue the list but I don't know what it should be≫

```
}
```

Fortunately for him, he knows his TA can read back what he is saying, so he puts a coded message in there for him.

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
```

≪And I know there's another to continue the list but I don't know what it should be≫
≪I guess I can leave it out for now and my T A will understand what I meant to write≫

```
}
```

The student then completes, to the best of his abilities, the functions in the LinkedList (`head` and `tail`) data structure and turns in his assignment to the TA.

At this point, the TA needs to grade this student's programming assignment. He runs the automatic testing suite, and finds that this student's program has failed all the tests. "How is that possible?" the TA thinks to himself. "He was doing OK in the beginning of the term."

Using Harmonia, the TA loads up the student's program. He reads the voice comments in the code and does not understand what the student meant to write. Perhaps if he played back the edit history of the document, he will be able to figure out where the student went wrong. Using Harmonia in XEmacs, the TA types in `M-x replay-history`. The document refreshes to an

empty state, and replays each edit at a rate of one every three seconds. When the computer replays a voice comment, it plays back the audio of the comment in real-time synchronized with the text edits that occurred while the student was speaking.

Once the edit history plays back the last voice comment, the TA understands that the student knew he should extend his LinkedList with a pointer to the next LinkedList in the chain (or nil). The TA then adds his own voice comment to the student's code:

≪The assignment says to create a linked list. I guess I'll need to declare the data structure.≫

```
public class LinkedList {
```

≪Well, there's definitely one slot for the value≫

```
    public Object value;
```

≪And I know there's another to continue the list but I don't know what it should be≫

≪I guess I can leave it out for now and my T A will understand what I meant to write≫

```
≪Yes, I figured it out.  You need to declare an-
other field with the linked list type.  Also try
to think about what you would do to point to the
end of the linked list.≫
}
```

The TA sends back the annotated assignment for the student to correct. He was able to provide the voice comment facility to give appropriate and more directed feedback to the student to help with his next revision.

9.2.2 Code Review

In this scenario, an employee of a networking startup in Silicon Valley is reviewing the code for a proxy server written by a colleague. He is reviewing the code at 2am because he is a night owl, and could not find a time to meet with his colleague that was suitable to both.

The code looks like this:

```
for (int i = 0; i < 10; i++ ) {
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
  execute(input, i);
}
```

First, the employee states the obvious:

≪It's a loop of 10 connections.≫

```
for (int i = 0; i < 10; i++ ) {
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
  execute(input, i);
}
```

Then, he notices an inefficiency. The employee's colleague is allocating a new Server object for every new connection when he should be reusing it:

≪It's a loop of 10 connections.≫

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

≪Why are you allocating a new server socket every time?≫

```
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
  execute(input, i);
}
```

Reading onward, he notices an egregious security violation:

≪It's a loop of 10 connections.≫

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

≪Why are you allocating a new server socket every time?≫

```
  Server serv = new Server(i);
  Socket sock = serv.recvConn();
  String input = sock.readStream();
```

≪@#@%! This is a huge security hole right here! You didn't check the input for validity before executing it.≫

```
  execute(input, i);
}
```

After making these comments, the employee quickly fires off two emails – one to the security officer at the company to turn off the alpha version of the server, and the second to his colleague berating him for leaving such as glaring security hole in their software.

This use of voice comments illustrates the benefit of informal voice commenting to annotate production source code and to rationalize quick decisions which everyone else can easily verify.

9.2.3 User Model

We next designed the user model. We integrated the voice commenting feature into Harmonia-Mode [37], an XEmacs plug-in developed for Harmonia. Harmonia-Mode uses the features of Harmonia to support interactive error detection and display, syntax highlighting, indentation, structural navigation and selection, structurally-filtered searches, and elision throughout the code document.

Our desired user model would allow the user to talk modelessly and have his speech inserted into the document at various points as voice comments. Some design questions about the user model concerned us.

1. **If a user speaks a voice comment, where should it go?** Should it be inserted at the current cursor position? Perhaps we should speech-to-text the comment and use artificial intelligence to interpret what he is talking about. For example, the user says "This field needs to be

renamed." before a class definition. Since the user is talking about a field, the comment could be associated with the field declaration. Could a comment be associated not just with a text position in the buffer, but also with structural elements in the syntax tree? This way a comment could be associated with a class, or a field, and even if that field is subsequently edited or moved, the comment could be kept near it. We decided to go with option 1, where the comment is inserted at the cursor position because it was the simplest option. Artificial intelligence is not our specialty, so we ruled out option 2. We ruled out option 3 as well because the text-oriented (as opposed to syntax-oriented) user model of the editor does not preserve enough information to determine anything but the text location where the comment should go.

2. **If a user speaks for a length a time while simultaneously editing the document, where does the comment go?** It could go at the position where he initially started speaking, or go at the end where he stopped speaking. Or the comment could span the entire range of characters. Unfortunately, Harmonia-Mode associates text ranges in the buffer with nodes in the syntax tree. Without a node to represent this expanded range, we could not make the proper association. We decided to insert the comment at the cursor position where the user first began to speak, since that was likely close to what he intended to comment.

3. **When should comments be inserted in the code?** Should they go in as soon as the programmer stops speaking, or should we wait a few seconds? If the comments go as the user is coding, it could disrupt the visual flow of the program and interrupt the programmer's thought process – exactly the opposite of our intentions. In addition, if the comment is too colorful for code (such as the expletive uttered by the code reviewer in the second scenario above), the speaker may not want it to go on. Likewise, the speaker may have forgotten to turn off the microphone when talking to a colleague in the room and the comment inadvertently was recorded. We could batch up spoken comments with their insertion locations and store these in a buffer. When enough time has passed or there are too many voice comments in the buffer, we interactively ask the programmer whether it should be inserted. It is important to present the voice comment inserted into the buffer with context above and below in order for the programmer to remember what they were thinking when they spoke the comment.

4. **How should voice comments be rendered?** Should a voice comment be translated into the programming language's syntax for a comment? If yes, the user would not be able to visually

identify which comments contained audio and which did not. We thus chose to present the voice comment in a distinct typeface. Should a voice comment be a simple one-character glyph or should we present the entire speech-to-text translation with special delimiter glyphs? We chose to render the entire text of the comment in order to facilitate skimming. If all of the voice comments needed to be played back in order to find out what was in them, then one would have to listen to all of the comments to find anything. To reduce the clutter from the voice comments, we enable the user to elide their visual representation into a two character glyph (the boundary tokens of the fully expanded voice comment). The user can also click on the voice comment and choose to have the recorded audio spoken back out the speakers.

5. **Can voice comments be edited? If so, are they editable in text or solely in audio?** If voice comments are modifiable via text, it is important to keep the audio in sync. One would have to know the time indices of all of the words in the audio stream and be able to cut and paste them. If the user reordered the words in the comment, the audio could be recut to match. But what if a user deleted a few characters of an existing word, or even added a completely new word? Should the tool synthesize new speech and insert it in the audio? Since this project was more about how the voice comments would be used as an annotation tool and not intended to be a complete prototype, we did not allow any editing of voice comments.

We made all the aforementioned modifications to our Harmonia-Mode for XEmacs (all except that code comments are automatically inserted into the code buffer two seconds after the programmer has finished speaking each one) and informally tried out our prototype. Barring some initial technical difficulties with IBM ViaVoice related to discovering when the user has started and stopped speaking into the microphone (the speech recognition engine only reports when it is decoding voice into text, and reports the overall input volume level, which the author used to infer when the programmer stopped speaking), the prototype worked well.

9.3 Experiments

The next step would be to conduct experiments to see how voice comments can be used by both novice and expert programmers to create better comments in their code. For novices, we could deploy the system to several students in an introductory programming class (conducted in the Java programming language), and ask them to use voice commenting on one of their programming

assignments. We would hope to see an increase in commenting in the program itself as compared to the rest of the students in the class.

Our main metrics will be the total number of comments, number of comments per class, field, method and line in the code, as well as the number of characters in the comments themselves. Another metric will be the number of comments that are about semantic information in the program, rather lexical and syntactic. This would be mainly a comparative one with the other students in the class, since novices tend to comment poorly anyway. Indirect metrics would be evaluations from the students as to the distraction or benefits they see from talking about their code, and an evaluation from readers who grade the programming assignments to see if they feel that they gain a better understanding of what the students were thinking when they were writing their programs. This last evaluation is critical to understanding whether or not playback of the comment audio is useful to the reader, as well as if it is possible to wade through the (hopefully) copious quantity of comments in the code to find out what is important.

For experts, we can conduct a similar experiment except that instead of a programming assignment, the expert's current project will be studied. A tool created by David Marin [65] can report statistics on commenting behavior found in a code repository. We could use this tool to establish a baseline for an expert's tendency to comment her code. Then, we encourage the expert to use our voice commenting system, and look at the changes, if any, in her commenting behavior by examining the repository. We can use the same metrics as for the novices as well as the same evaluation by the programmer to identify any cognitive issues that interfere with the programming activity.

9.4 Summary

We hope further experiments will show that programmers, both novices and experts, will be able to use this tool to comment their code more completely and descriptively. Exploration of the cognitive interference issues will be critical to its success – speaking about one's code may interfere with programming, or even if not, the stream-of-consciousness style of comment may increase the likelihood that the comments will be lexical or syntactic in nature, rather than the more useful semantic form of comments.

The one incontrovertible benefit of voice comment is that it will form an indelible and more complete record of the programmer's process while performing his duties. Programmers have a lot to say – it is time that we started capturing it.

Chapter 10

Conclusion

This dissertation has proven our thesis:

1. Programmers *can* learn to program using voice recognition.
2. It *is* possible to successfully understand spoken program code and commands and render them as code in a traditional text-based form by adapting a program analysis system for spoken input.

This dissertation's contributions follow a human-centric view of the problem. First, we studied programmers to understand how they would speak code and navigate through documents independent of any system that might have to understand them. We used this information to design a spoken input form that balances a human's desire to speak in natural language with our ability to develop a speech understanding system to analyze it. Then, we implemented this speech understanding system by taking existing program analysis tools that were created for unambiguous programming languages and adapting them to understand ambiguous speech. We plugged these new analyses into a program editor and then tested it with professional programmers to see how well both the programmers and the system perform. The lessons we learned will help us design the next version of the system as well as advance the community's knowledge about how programmers express themselves when programming.

This is the concluding chapter of this dissertation. We begin by discussing what we learned by designing and building these algorithms and systems and objectively evaluate the work. We lead into a discussion on structure-based editors and consider their future when operated by voice. We then present future work and illustrate the many different directions in which program-

ming by voice may go. Finally, we end with a discussion of where the field is today and where it could go in the future.

10.1 Design Retrospective

There are several major components to this dissertation.

1. **Spoken Program Studies** – Studies of programmers to identify how they would speak code and navigation commands.
2. **Spoken Java Language** – Design of the Spoken Java language.
3. **XGLR Parsing Algorithm** – Design and implementation of the XGLR parsing algorithm.
4. **Inheritance Graph** – Design and implementation of the Inheritance Graph data structure for names, scopes and bindings.
5. **SPEED** – SPEech EDitor composed of Shorthand, Eclipse, What Can I Say?, How Do I Say?, Cache Pad, Context-Sensitive Mouse Grid, and a plugin for a speech recognizer.
6. **SPEED User Studies** – Studies to find out how developers learn to program by voice.

The following sections discuss each component above. We end with a overall review.

10.1.1 Spoken Program Studies

There were two programmer studies conducted in the early days of this research. The first looked at how programmers might speak code out loud if they read it off a piece of paper. The second looked at document navigation commands in commercial speech recognizers.

In the first study, we learned a lot about the ambiguities present in speech that cause problems for understanding spoken programs. Homophones, punctuation, spaces between words, and capitalization all cause difficulty for automatic recognition of spoken code. Abbreviations and partial words are difficult to say out loud, and appear quite often in programs, especially in system libraries. In hindsight, many of these issues seem obvious, but there had been no studies before ours detailing the issues.

Programmers tend to speak in abstractions when they perceive patterns in their code. It is not clear yet whether this is a side-effect of speaking pre-written code or something that programmers would also do when speaking spontaneous code.

In the SPEED user study session that employed commercial speech recognition tools, we noticed that programmers tended to speak in short bursts due to recognizer malfunctions. Sometimes the wrong word was recognized, but more often, no words were recognized. When the wrong word appeared in dictation, programmers had to spend time and commands to correct the error. This effect was not seen in the spoken programs study because participants were programming into a tape recorder, and could not see the results of the transcription.

The document navigation study showed that all forms of document navigation provided by commercial speech recognition are sub-par. They take too long to activate and compared with mouse and keyboard navigation require too much reading and speaking. The biggest problem exposed by watching people navigate documents by voice is the cascading recognition error. People who tried to use the find dialog box would speak words that were incorrectly transcribed. When they tried to undo their utterance by saying "scratch that" or "correct that," the recognizer would write down those words instead of interpreting them as commands. Subsequent utterances might be interpreted correctly or incorrectly, but usually it proved easier to exit and restart the dialog box to begin with a clean slate.

10.1.2 Spoken Java Language

Spoken Java was designed to be natural for experienced Java programmers to speak out loud. Since most programmers speak the same words when speaking Java code, it sufficed to build a single lexical specification and grammar for the language. There is plenty of variability supported in the language. For each lexeme, there are often three or four ways to say it. For several grammar constructs, there are two or three ways to say them, including dropping optional punctuation markers (or their equivalent verbalized forms). Our SPEED user study showed that developers who were trained to speak the English words in their program without the punctuation felt that they understood exactly how to enter the code in the acceptable form.

While this design aimed to cover the majority of the spoken form of Java, it is not and cannot be complete. A robust system would need to be extensible, and learn how the particular programmer verbalizes code. Learning would work fairly well because even though programmers exhibit a fair amount of variation in the words they choose to use when speaking code, it is drawn from a finite vocabulary and grammar. It should be possible to adapt natural language grammar machine understanding systems to making the Spoken Java language specification extensible.

In the SPEED study, several developers felt more comfortable with code template in-

sertion commands rather than directly dictating code. They explained this by associating code templates with higher-level coding actions, describing code rather than speaking it literally. Our Spoken Programs study also saw this trend during verbalization of code read from a piece of paper. Programmers tended to abstract the code on the paper, especially when there were obvious patterns. There were no obvious patterns in the SPEED study, but programmers still wanted to use abstractions. Supporting user-identified program abstractions will be an important topic for future study.

10.1.3 XGLR Parsing Algorithm

The XGLR parsing algorithm is without a doubt the highlight of the technology developed for this research. It is the only parser capable of handling lexical ambiguity, not just when one token has multiple spellings, but also when a single token has multiple meanings (lexical types) *and* when a character stream may have several different interpretations leading to a distinct sequence of tokens being produced for the parser to consume. Further parser developments are ongoing in the areas of structural editing (using single tokens to represent subtrees) and parsing program fragments. Both of these features require significant alterations to the parse tables that can be done only using Generalized LR parsing as a substrate.

As part of the XGLR project, we developed Blender, a combined lexer and parser generator. Blender is composed of two languages modules, one representing the lexical specification and one representing the grammar of the input files. Each is written in Blender itself. Blender is bootstrapped from language modules written with Flex and Bison. Once the zeroth Blender is loaded, its additional power is used to compile the language modules for the first Blender. Further research in whitespace specification and language embedding should enable us to create a second Blender, developed with the increased power of the first, which will support a lexical specification composed of three sub-languages, file structure, regular expressions and C. This composition would also support a simpler file format, approaching the simplicity of the original Flex file format, which employs whitespace as language boundaries between sub-documents.

Blender's parser generator uses DeRemer and Pennello's LALR(1) lookahead set generation algorithm. This algorithm is extensively referenced in the literature, but is almost never implemented. There are relatively few parser generators in the world, and most people who need to write a language use one of the existing generators to do it. Bison contains an implementation of this algorithm, but does it all with data structures that contain only two and three letter variable

names used in data structures that consist solely of varieties of C integer arrays. There is almost no documentation and no high-level data structures.

Given that it has been 23 years since the publication of the paper, much of the terminology used within has evolved. Since most compiler students never create their own parser generator, most programmers today are completely unfamiliar with the knowledge required to implement this algorithm. Consequently, it was found to be such a challenge to implement DeRemer and Pennello's algorithm that we felt compelled to include a concrete easy-to-understand, easy-to-copy implementation in Appendix D of this dissertation. DeRemer and Pennello's algorithm is clever and well-explained, but only if you already understand it. Since it is hard to understand, it is easy to have bugs. We would like to explore ideas for validating generated parse tables for large languages in which manual table validation proves infeasible. One bug in our parse table generator survived a whole year of use before being discovered.

There is considerable speculation as to how much benefit parsing embedded XGLR languages might receive from moving to scannerless parsing. GLR support for lexing would provide the needed grammatical mechanisms and structures for input stream ambiguities – in fact, the results of our parse are very similar to what you might get with a scannerless parser. We still feel, however, that a scannerless approach loses some useful functionality with respect to parser incrementality and lexical specification. Having a separate lexical phase prevents simple changes to the spellings of variably-spelled tokens (e.g. identifiers, numbers) from causing the parser to reanalyze the edit. Scannerless parsing also suffers from extra overhead with parser conflicts on tokens with common prefixes (keywords vs. identifiers). It has no longest-match regular expression criterion and no ability to prioritize tokens by their order in the lexical description. While scannerless parsing systems can employ disambiguation filters to approximate these facilities, their deployment is not as simple or easy to use as in the separate lexical specifications.

XGLR was implemented by extending an incremental parsing framework. It was believed at the time that this framework was created (mid 1990s) that batch parsing a document after every keystroke edit would take too much time and destroy the user interface experience. For the most part this is not true today. With CPU speeds in the multi-gigahertz range, a batch parse of a 20K file takes around 30-50 milliseconds. The incrementality in incremental parsing adds additional overhead compared to batch parsing, so even though a program edit may have small effects on the parse tree, it may still take on the order of several milliseconds to process. This overhead could be mitigated by increasing the granularity of the incremental parse to the current method or structural unit. The added complexity of maintaining the parser's incremental features was quite painful and

caused many bugs until we got it right.

Implementation issues aside, incrementality does have its benefits. Incrementality enables the parser to preserve annotations on the unchanged portions of the parse tree, as well as identify the structural portions of the tree that have changed. This is a very good substrate upon which to build a semantic differencing engine. The difference between two parse trees is simple to extract from our versioned parse tree data structure. Processing this syntactic difference is the work of a fairly routine and automatically generatable semantic differencing algorithm. Semantic differences can be used for documentation of changes to source code, for automatically coding a user's edits at a level higher than keystrokes and mouse motions, and for supporting structural undo operations that liberate the user from the chronological jail of traditional undo implementations.

10.1.4 Inheritance Graph

The Inheritance Graph simplified the scoping data structures used in Java semantic analysis. By storing all of the type and use-def information in a separate data structure, later analyses could reference all of this information without reconstructing it from scratch. Type checking for most languages is fairly easy to implement using the Inheritance Graph since it can report the type of a reference knowing only the reference's name and position in the program.

Unfortunately, for Java, the Inheritance Graph proved to be a big disappointment both in terms of power and utility. The formalism is not powerful enough to describe Java or C++, languages with context-sensitive name visibility rules. The way the name appears in the program affects the name lookup algorithm, obviating the IG's language-independent algorithm for name propagation. Consequently, the Inheritance Graph for Java or C++ cannot quickly and efficiently answer the questions, "what does this name mean?" and "what names are visible here?" Fast, efficient access to this information was going to be the lynch pin of semantic disambiguation, the technique for taking a forest of parse trees and deducing the ones that are semantically correct in the context of an existing program. This problem does *not* arise for languages without context-sensitive visibility, which covers almost all other programming languages including C, Lisp, Pascal, Cobol, and many others. Hence, if one were to build programming by voice solutions for other programming languages, this approach might prove useful. Further study would be needed to find out.

As we attempted to use the Inheritance Graph for disambiguation, we noticed that disambiguation and type checking shared many of the same algorithms, especially for name lookup. One

could either take each terminal in a parse forest and look it up to see what kind and type it has in order to rule out illegal interpretations, or enumerate each interpretation, insert each one into the parse tree and type check the parse tree. It turned out to be much more economical to take the type checking approach. This might appear expensive, due to the many interpretations found inside of an ambiguous subtree, but it is not slow in practice. Most ambiguous parse forests have less than ten interpretations, and Java type checking is fairly quick when run on a single compilation unit.

All this being said, it turns out that even this form of disambiguation is not useful in practice for Java. Each ambiguous interpretation is often so far removed from the others that a human presented with a list that holds as many as 20 ambiguities can spot the correct interpretation with very little time or mental effort. In addition, since programs may be written out of order (in other words, not top down) it is fairly common to use names before they are defined. If strict type checking rules are used, those constructs are illegal. If they were filtered out completely, and they were the ones that the user intended, then the user would be unable to choose the right interpretation when he wanted to. Likely, the user would believe there to be a bug in the analysis system and try to redictate the offending code. It is better to let the Java programmer have his way and accept semantically illegal (but syntactically legal) code than to worry about presenting him with too many interpretations from which to choose.

10.1.5 SPEED

The SPEED editor is a composition of three main components: Eclipse, Shorthand, and the speech recognizer. Eclipse is an unwieldy substrate upon which to base an editor. Eclipse's developers say it was designed for extensibility, but this really only holds in the design of the core APIs. The Eclipse Java Development Toolkit (JDT) is a completely custom, one-off creation and was not designed to be very extensible at all. In creating Shorthand, we had to modify several pieces of JDT source code to expose the requisite APIs. We even had to disable the ability of the JDT to auto-load files with the `.java` extension to prevent it from activating in place of the Shorthand editor.

Shorthand turned out to be a structure-based editor, despite our intentions to have it behave in a more freeform manner. In order to enable any of its editing features, it needed to have access to the parse tree of the document. In fact, Shorthand requires that the parse tree be maintained correctly, or its commands fail to work properly. The history-based error recovery created by Tim Wagner for his incremental LALR(1) parser [104] and subsequently used by our XGLR parser

implementation is intended to preserve as much of the parse tree structure as it can in the face of erroneous inputs and edits to the program. However, the structure returned by this error recovery *has* errors inside. Within the erroneous subtree isolated by error recovery, parse tree information is suspect, and certainly the terminals near the edit site are incorrect. Thus, Shorthand, a client of the parse tree, must rely on incorrect information to run its user interface. To make sure this never happened, we provided only syntactically valid structural editing operations to the user, in the spirit of the earliest, strictest, structure editors.

This is not all bad, however. Within the limits of structural edits, there can be surprising amounts of flexibility. The editing facility for voice allows for freeform dictation bounded by a target nonterminal to which the input must conform. When there is a series of chain productions in the grammar, we choose the highest nonterminal in the grammar to provide the most flexible interpretation of the input.

Another problem we found with Shorthand was in code authoring. When code is written by the user one token at a time, it usually does not parse correctly until the last brace or semicolon is inserted. This means that a structure-based editor will have incorrect structure in the area where the programmer needs it most – where he is currently writing code.

To try to work around this difficulty, we created a novel predictive parser called a program fragment parser [9]. Built into a modified GLR parser framework, program fragment parsing can start a parse at *any* state in a parse – in fact, it uses the first word entered by the user to choose parse states in which to start. The parse proceeds normally until the first reduction. Normally on a reduce action, the parser would traverse the graph-structured stack backwards by the number of symbols in the production being reduced. However, if the parse started in the middle of the production, there are not enough graph-structured stack nodes to traverse. If the parser reaches the beginning of the parse before having traversed enough nodes, nodes are created to fill out the beginning of the production – nonterminal completer nodes and terminals. Likewise, if the parse input runs out before the parse is complete, symbols are created on the right side until the parser can reduce to a nonterminal whose yield covers the entire actual input. This strategy creates a forest of parse trees for all possible structures the user could have been typing, which can be used to reenable structure-based editing. Since the structure is now ambiguous, care must be taken to ensure that any available actions are consistent with all interpretations of the input. As the user enters more words, the parse trees become more and more constrained, until a single interpretation is reached, and the system can completely predict the structure that the user is typing.

Program fragment parsing is a nice idea, and useful for many more applications than

interactive editors. Code fragments are quite common, appearing in email, instance messages, software developers' spoken conversations with one another, and in programs written in embedded languages. It is useful to be able to analyze these fragments of code in isolation, especially when the surrounding program context is missing or difficult to analyze.

The difficulty with program fragment parsing, however, is that the degree of ambiguity found in typical programming languages is more than our implementation can handle. With short phrases of up to three words, there are often thousands of possible parses. This is mainly due to the recursivity of most programming language grammars, especially at the statement or expression level. The problem we encountered is very similar to the situation natural language parsing experts found themselves in in the 1960s when trying to parse English sentences. There was so much recursivity and variable sentence structure available that there were too many parses to handle. In the end, we abandoned program fragment parsing, and reverted back to a pure structure-based editing strategy.

Spoken Java's command grammar is not as powerful as it could be. We had hoped to integrate the command grammar with the language commands in order to support editing, navigation and transformation operations that referenced fragments of code. Instead, we created a separate command grammar using the speech recognizer's support for finite-state command grammars. Whenever code can be entered, SPEED switches into dictation mode and matches the more free-form input against the Spoken Java language directly. This engineering choice is not too much of a limitation given the fairly modest capabilities of this first prototype of SPEED. A structure-based editor lends itself to small self-contained operations on small fragments of the program; the ability to speak arbitrary code in the command is less necessary when there is adequate support to refer to these pieces of code through the user interface (for example, through mouse grid). It would be useful, however, to support Dragon NaturallySpeaking's Select-and-Say operation on code. Avoiding the use of dictation mode for commands should result in more accurate recognition of those commands. Commercial speech recognizers are highly tuned for command mode recognition; they are much less accurate in dictation mode.

Cloudgarden's JSAPI implementation works as advertised, but fails to integrate with native speech recognition user interfaces. In fact, if Dragon NaturallySpeaking is running, Cloudgarden's JSAPI often cannot start up. When it does, it misleads the programmer by claiming to work properly, but then simply will not recognize anything the user speaks into the microphone. All of Dragon's extensive built-in grammars are turned off as is the entire Dragon user interface, which was carefully designed and built over eight generations of the product. We had to reimplement many

of their features using JSAPI. Aside from this, the voice recognition interface for SPEED turned out to be very simple. Shorthand defines actions using keyboard or voice inputs. We gather all the voice inputs into various rules in a speech grammar and activate them. Whenever a command is recognized, the text of the command is sent directly to Shorthand for processing. While recognizers often claim to support changing grammars and rules after each word or command recognized, this is not quite as true as advertised. Changing grammars or rules (even to just turn them on and off) pauses the recognizer for up to one second. An early prototype of SPEED activated and deactivated grammars for words that were available based on where the user had clicked. However, recognition of multiple commands in a row was frustrated due to the delay in changing grammars after each individual command. The subsequent commands were never heard.

Context-sensitive mouse grid is a success. It is fairly easy to use, and precise in its ability to point to any interesting program structure. In fact, the current version of Dragon NaturallySpeaking contains a context-sensitive mouse grid numbering the links on a web page in Internet Explorer. If more of the programming task could be rendered as choosing numbers from a list, the programming tasks would become more efficient and less prone to inaccurate transcription by the voice recognizer. The structural parts of a program (as indicated above) are good candidates for this kind of editing since there are very few structural elements (fitting into a small numbered list) and these elements have few truly variable forms, other than their label given by the user. Imperative code found inside method bodies and field initializers is less amenable to this approach because of its highly variable nature.

Cachepad suffers from a similar problem as the What Can I Say? view, namely too many items to display. There are many unique identifiers found in Java programs, and the cachepad can only hold 20. Cachepad uses plugin policies to determine which ones go in, but we have not yet figured out the "right" policy that comes up with a cachepad full of identifiers that programmers are likely to need. We plan to do a study of edit logs produced by programmers editing code over significant periods of time to see if we can produce an oracle that can predict the next ten identifiers that will be used at any point in the log. This oracle, if it can be discovered, will be the "right" policy.

10.1.6 SPEED User Studies

When we started conducting the SPEED user study, we intended to use only commercial speech recognizers. However, once the initial three participants used the software, we realized that

the extravagant error rates were impeding our ability to understand how SPEED could be used. Consequently, we employed a human speech recognizer to take the place of the recognition software. This recognizer's near-flawless accuracy enabled participants to get much further in writing a small Java program and let them use many more features in SPEED. While a human recognizer does not represent the state-of-the-art in commercial or research speech recognition software, it does indicate where the speech recognition field may end up with another burst of progress.

Users learned the commands very easily and made few errors on their own. Most errors were caused either by the speech recognizer's inaccurate transcription or by bugs in the SPEED editor present at the time of their participation in the study. Users did want to customize the commands to make them easier to say and more familiar to what they were used to. If this feature were not available, they all thought it would only take at most a week of training to become fluent in the supplied command vocabulary and grammar.

Code template insertion was used far more often than code dictation (outside of editing identifier names). Programmers perceived dictation to get "more bang for the buck." This result concurs with our own GOMS analysis, as well as with Snell's findings that code template insertion is a useful feature for programmers to most easily enter large amounts of code. In addition, participants felt that describing code with code templates was a higher level means of programming than code dictation. While they might type code literally, speaking code literally was more foreign to them.

None of the participants would use this software unless they had to due to their circumstances, such as contracting RSI or being in a hands-free situation. However, we found from our pre-study interview that programming was not a significant component of their daily workday, leading all of them to the conclusion that the use of a voice-based programming tool, while inefficient compared to keyboard, would not slow them down significantly in their job performance.

10.2 Structure-based Editing by Voice

A recent study showed that Java developers typically perform structural operations when they program, even though their editors support only textual edits [54]. What is not clear from this study is whether programmers understand that they are performed structural edits either before or after they make the textual edits. Certainly afterwards, a programmer can spot the structural edits (e.g. changed a method name, inserted a new variable, changed the value of the argument to a procedure call) they did, but what about before? Much of the early research in structure-based editors came with the assumption that programmers could not only intuitively understand how to

make structural edits to their programs, but that they would want to think of their edits that way. Alas, there is no research to explore this issue; in fact, it may be quite difficult as even a think-aloud protocol might perturb how the programmer perceives the changes they are making. In fact, it may be that editing by keyboard involves a form of motor memory and activity that may not be consciously controlled. By moving the user to structural editing, the environment may be removing their facility for keyboard-based editing below conscious thought and moving it up to conscious thought, increasing the cognitive load and likely slowing the programmer down. Of course, there is one other major reason why structure-based editors did not catch on: programmers are resistant to giving up text-based editing because of its powerful robustness in the face of lexical, grammatical and semantic errors that might occur in the program at any stage during the edit.

Another issue came up with Shorthand that surprisingly does not appear often in criticisms of structure-based editors. There are too many commands, and they are context-sensitive. In order to learn all of the the operations you can perform on a particular structure in the program, you would have to have a program that exemplified them all. This proves quite difficult in practice. Often, programmers encounter editing situations on a code construct they had never edited before, leading to confusion as to what operations they can perform and how to invoke them. The What Can I Say? and What Can I Type? views help, but the sheer number of commands available causes these views to scroll, leaving many commands invisible on the screen. It reminds us of DOS WordPerfect, which required a paper template to fit over your keyboard with all of the 50 function-key commands on it.

Structure-based editing is also about details, and a human's understanding of these details often runs contrary to the grammatical representation. For example, inserting a `final` keyword into a class, method or field declaration involves adding it to a list of modifiers, but placing it on a local variable declaration means adding it to an optional grammar production. You can add another field to a field declaration using the command 'insert after', but if you had intended 'insert after' to insert a completely new field declaration, you would have had to expand the selection from the field name to encompass the entire field declaration before using the 'insert after' command. Inserting new code templates requires getting the indentation and the carriage returns inserted into the right places. The only reasonable option here is to pretty-print the code for the user, potentially erasing important spatial landmarks in the code.

Another mismatch in perception between human and machine concerns whitespace. Harmonia's parse tree stores whitespace to the right of every terminal in the program. But a human sees whitespace as the space from the beginning of the line to the first terminal on that line, and the space

after the end of the last terminal on that line. Line boundaries in Harmonia are often buried inside of whitespace terminals, leaving the editor to parse line breaks without parser support. All of these are examples where the compiler-based grammar used in our program analyses created needless complexity for the editor.

Michael Van de Vanter warns that the edit-time grammar and the compile-time grammar of programming languages *should* be different because the needs of their client applications are different [18]. We concur. In fact, we will go one step further and say that the needs of the client application usually do not require perfect, difficult-to-implement analyses, but coarser, easy-to-implement analyses would suffice and be more robust. A compiler certainly needs strict adherence to a grammar and semantics in order to produce a correct binary. But an editor, even a structure-based editor, requires too much of a programmer if he must conform to the structure all the time. Advances in structure-based editors, including in this dissertation, enable limited free-form editing, but still require the structure to end up being correct after the edit is complete. This, too, is still confining.

A better structure for structure-based editors would be based on a coarse parsing technology. Eclipse's JDT contains seven parsers for various purposes. A simple implementation would have at least two independent parsers: one for the structure of the program file itself (class, method, field, etc) and another for imperative code (statements and expressions). Structural elements can easily be programmed in a top-down way. Imperative code is often written bottom-up, and only later contained within grammatically correct structures. Until the containment is complete, structure-based editors would either make this form of edit illegal, or fail to provide services on the text in this area. Instead, if a coarser parser were used to identify possible or probable containing structures, a limited set of services might be available. In addition, since containment, in Java indicated by braces, is often broken in the middle of text edits, robustness to this form of breakage is essential. Clues about containment can easily be gathered from indentation (similar to Python and Haskell), from style (variable declarations often appear at the beginnings of blocks, but not at the end), or not at all, when it does not affect the editor services. If a variable's lifetime is not properly bounded because of a missing close brace, including that variable in code completion will not cause problems for a human programmer who can easily ignore the inappropriate reference.

We think that the future of structure-based editors lies in statistical learning and inference techniques. Important features such as style of indentation, containment, variable name choice and length, common semantic constructs (such as always storing the return value of a system function in a newly declared local variable), name-type bindings, and even bugs can be extracted by studying

existing programs. This database of knowledge can be used to infer structure when precise parses and semantic analyses fail due to incorrectness or incompleteness. When parsing does succeed, but the semantic interpretations are still ambiguous, statistical inferencing can help assign proper probabilities to the choices, helping raise the likely choice to the top.

10.3 Future Work

Voice-based entry of code introduces many lexical and syntactic ambiguities that cannot be resolved until semantic analysis is run. In the system built for this dissertation, lexical and syntactic analysis phases must generate all possible interpretations of the input in order for semantic analysis to choose the correct one. In some cases, this process may not scale (as natural language researchers discovered about English language analysis in the 1970s). It should be possible to use partial parsing (based on our work in program fragment parsing with GLR) and partial semantic analyses to help prune ambiguities as early as possible in the analysis process. Additional techniques can be developed by adapting natural language disambiguation algorithms to the more limited domain of programming languages.

In addition to composing and editing code, the programmer may wish to perform high-level program manipulations. For example, she may invoke a program refactoring or search for a particular structural or semantic entity in the code base by saying "Find all references to the MyList dot getElement method and replace them with StandardList dot getElementAt". To support this combination of commands and code, we must use Blender to define a new embedded language of Spoken Java code embedded within a grammar for the SPEED commands, and create a semantic analysis that can understand the combination. This analysis would trigger either built-in or custom-designed transformations provided by the editor infrastructure.

Programmers do much more than create, edit and navigate through code. They also debug, document and review code written by themselves and by others. It is important to extend voice-based interaction to these tasks. SpeechClipse [90] and EmacsSpeak [80] are good solutions to this problem.

In the second session of our second user study, we employed a non-programmer human to transcribe the spoken commands and code of the programmer study participant. It would be fascinating to ask a third-party software developer to try to transcribe a spoken program. How often does a human programmer correctly understand spoken code? The places where a human has trouble ought to be especially difficult for a computer, if he were to employ similar algorithms

for speech understanding. Another interesting study would deliberately try to confuse the human transcriber by speaking odd programs that use non-obvious or ambiguous words to represent Java constructs. A similar study would be to look at how pair programmers understand one another when they speak code.

We did not do any study of editing prior to the SPEED study. While we are confident that the spoken language we discovered for the Java programming language is the one that many people speak, we do not know what kind of natural commands people would use to move around the editor. Since we designed the SPEED editing commands by fiat, without the help of a user study, we propose a Wizard of Oz study to look at this issue further. Set up a programmer in front of an editor with a microphone, but no keyboard and mouse. The user must speak commands and code into the microphone to program a simple data structure. The Wizard listens to the microphone and secretly interprets what the user says according to a loose set of rules, moving the cursor around and typing in code. The session and its code are recorded and coded for later analysis. It is not obvious that non-voice recognition users will have any novel ideas for navigation, but since the wizard can not react to any physical actions the programmer takes, the recording will have a complete record of all utterances required to get the program into the editor.

Choosing options from a numbered list is one of the most reliable voice interfaces. It is this author's intuition that if many more interfaces on the computer could be reduced to a list of numbered choices, then speech recognition would become much more usable. For programming purposes, top-level program structures in statically typed object-oriented languages (like classes, functions, modules, etc.) vary little in typical programs – usually the name is the only parameter that varies. While code template insertion can take care of these cases, the activation of the code templates could be done with a numbered list followed by the user speaking the name of the created construct. Once function body code needs to be entered, dictation support is warranted, due to its much more variant forms and larger number of options available at any program position.

A major component of this work was the design of a spoken variant of the Java programming language. This design was guided by studies of Java programmers speaking code. The design methodology we used could be applied to other programming languages, and even other formally specified languages, such as scripting languages, design languages, command languages and domain-specific languages. Languages that lend themselves to static program analyses would also be good candidates for our methodology and technology in supporting programming by voice.

10.4 Future of Programming-by-Voice

A major component of programming by voice is its dependence on commodity desktop speech recognition tools. Unfortunately, desktop speech recognition never grew larger than a niche market, leading several vendors to abandon their products in favor of back-end telephone call centers. As our user study of developers using SPEED showed, even these commercially available products are not yet robust enough to power programming by voice without either more high-quality, accurate recognizers or more speech training by software developers. We had thought that retraining the statistical database included with the speech recognizer would gain us more accuracy. However, the training material — the programs in the software developer's code base — is not sufficiently general to be useful. The identifiers defined in one program are either unique to that program, which means their inclusion in the training set skews the probabilities, or they appear in other programs in slightly altered form, leading to greater likelihood of recognition error. This problem will have to be addressed by further research in speech recognition.

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

10.5 Final Summary

Programming-by-voice systems can be a viable alternative to keyboard-based programming environments, especially for those suffering from repetitive strain injuries. By first learning how programmers naturally verbalize code and then developing a formal spoken code analysis system based on the lessons we learned, we are taking one of the first human-centric approaches to achieving the goals of this field. Our studies revealed valuable information about the kinds of ambiguities that emerge from spoken programming (which do not appear when using a keyboard), about the use of voice expression and prosody for disambiguation, about the differences between native English and non-native English speakers, and about the human tendency toward abstraction over verbalization of details.

Based on these initial lessons, we designed a new dialect of Java, called Spoken Java,

which is easier to speak out loud. We have used programming language tools to formally describe Spoken Java, and have enhanced these tools to support the kinds of ambiguities that arise from spoken programs. We classified the lexical ambiguities caused by these situations into four types, and developed both a lexer and parser generator and a set of lexing and parsing analysis enhancements to address each one. These tools enabled us to create a program editor that supports programming by voice for three kinds of tasks: code authoring, editing and navigation. These tools have been incorporated into the SPEED editor, which we have evaluated through a user study. We found that programmers are able to learn to program verbally with small amounts of practice, but have significant trouble when the speech recognizer misinterprets their commands and dictation. Programmers prefer high-level abstraction to code dictation, and perceive voice-based programming to be less efficient than keyboarding, but efficient enough to perform their daily work adequately.

Programming-by-voice can enable motor-impaired software engineers to continue programming, albeit at reduced efficiency compared with an unimpaired programmer. With more study, different user interface designs and better analysis tools, software developers will one day be able to use voice-based programming to compete effectively in the workforce.

Bibliography

[1] Johnny Accot and Shumin Zhai. Beyond Fitts' law: models for trajectory-based HCI tasks. In *Proceedings of ACM CHI'97 Conference on Human Factors in Computing Systems*, pages 295–302, 1997.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[3] Alexander Aiken. Cool: A portable project for teaching compiler construction. *SIGPLAN Notices*, 31(7):19–24, 1996.

[4] Stephen C. Arnold, Leo Mark, and John Goldthwaite. Programming by Voice, VocalProgramming. In *Fourth Annual ACM Conference on Assistive Technologies*, pages 149–155. ACM, 2000.

[5] John Aycock. The design and implementation of SPARK, a toolkit for implementing domain-specific languages. *Journal of Computing and Information Technology*, 10:55–66, 2002.

[6] John Aycock and R. Nigel Horspool. Schrödinger's token. *Software—Practice and Experience*, 31(8):803–814, July 2001.

[7] Marla J. Baker and Stephen G. Eick. Visualizing software systems. In *Proceedings of the 16th International Conference on Software Engineering*, pages 59–67, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

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

[9] Andrew Begel and Susan L. Graham. Parsing program fragments with GLR. Submitted for publication, 2005.

[10] Andrew Begel and Susan L. Graham. Spoken programs. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, September 2005.

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

[12] Daniel Brantley and David Dillard. Software tools in the service of documentation. In *Third International Conference on Systems Documentation*, pages 60–70, 1984.

[13] Angela Carbone, John Hurst, Ian Mitchell, and Dick Gunstone. Principles for designing programming exercises to minimise poor learning behaviours in students. In *Proceedings of the Australasian Conference on Computing Education*, pages 26–33. ACM Press, 2000.

[14] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum, Hillsdale, 1983.

[15] Kevin Christian, Bill Kules, Ben Shneiderman, and Adel M. Youssef. A comparison of voice controlled and mouse controlled web browsing. In *Proceedings of the Fourth International ACM Conference on Assistive Technologies*, pages 72–79, 2000.

[16] Cloudgarden. TalkingJava SDK. http://www.cloudgarden.com/JSAPI.

[17] Robert P. Corbett. *Static semantics and compiler error recovery*. Ph.D. Dissertation, Computer Science Division—EECS, University of California, Berkeley, CA, June 1985.

[18] Michael L. Van de Vanter. Practical language-based editing for software engineers. In *International Conference on Software Engineering Workshop on Software Engineering and Human-Computer Interaction*, pages 251–267, 1994.

[19] Frank DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

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

[21] Françoise Detienne. *Software Design – Cognitive Aspects*. Springer, 2001.

[22] Charles Donnelly and Richard Stallman. *Bison: the Yacc-compatible parser generator*. Free Software Foundation, December 1990.

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

[24] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[25] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph — static and dynamic graph drawing tools. In Michael Jünger and Petra Mutzel, editors, *Graph Drawing Software*, Mathematics and Visualization, pages 127–148, Berlin, 2004. Springer-Verlag.

[26] Timothy E. Erickson. An automated FORTRAN documenter. In *Proceedings of the international conference on systems documentation*, pages 40–45, 1982.

[27] Mikael Ericsson, Magnus Bauren, Jonas Lowgren, and Yvonne Waern. A study of commenting agents as design support. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems (Summary)*, volume 2 of *Late Breaking Results: Support for Design: Experiments and Cybertools*, pages 225–226, 1998.

[28] R. Escalona. Case study of the methodology of J. D. Warnier to design structured programs as systems documentation. In *Third International Conference on Systems Documentation*, pages 95–100, 1984.

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

[30] Stuart I. Feldman. Implementation of a portable Fortran 77 compiler using modern tools. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, pages 98–106, New York, NY, USA, 1979. ACM Press.

[31] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(22):381–391, 1954.

[32] Joan Francioni and Ann Smith. Computer science accessibility for students with visual disabilities. In John Impagliazzo, editor, *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, volume 34, 1 of *SIGCSE Bulletin*, pages 91–95, New York, February 27– March 3 2002. ACM Press.

[33] Phillip Garrison. *Modeling and implementation of visibility in programming languages*. Ph.D. Dissertation, UC Berkeley, 1987.

[34] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference of Software Engineering*, pages 645–654, Edinburgh, Scotland, UK, May 2004. IEEE Press.

[35] Susan L. Graham, William N. Joy, and Olivier Roubine. Hashed symbol tables for languages with explicit scope control. In *SIGPLAN Symposium on Compiler Construction*, pages 50–57, 1979.

[36] Harmonia Project Web Site. http://harmonia.cs.berkeley.edu.

[37] Harmonia Research Group. *Harmonia-Mode Project Documentation*. http://harmonia.cs.berkeley.edu/harmonia/projects/harmonia-mode/doc/index.html.

[38] X. Huang, A. Acero, F. Alleva, M.-Y. Hwang, L. Jiang, and M. Mahajan. Microsoft Windows highly intelligent speech recognizer: Whisper. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 93–96, Detroit, MI, May 1995.

[39] IBM. ViaVoice Speech Recognizer. http://www-3.ibm.com/software/speech/.

[40] Takeo Igarashi and Ken Hinckley. Speed-dependent automatic zooming for browsing large documents. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 139–148, 2000.

[41] Takeo Igarashi and John F. Hughes. Voice as sound: Using non-verbal voice input for interactive control. In *Proceedings of the 14th Annual Symposium on User Interface Software and Technology*, pages 155–156, New York, November 11–14 2001. ACM Press.

[42] F. Ives. Unifying view of recent LALR(1) lookahead set algorithms. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 131–135, Palo Alto, CA, June 1986. ACM.

[43] Fred Ives. Response on remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, 1987.

[44] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, 1996.

[45] Bonnie E. John and David E. Kieras. Using GOMS for user interface design and evaluation: which technique? *ACM Transactions on Computer-Human Interaction*, 3(4):287–319, 1996.

[46] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[47] Clare-Marie Karat, Christine Halverson, John Karat, and Daniel Horn. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Speech and Multimodal Interfaces*, pages 568–575, 1999.

[48] Azfar S. Karimullah and Andrew Sears. Speech-based cursor control. In *Fifth Annual ACM Conference on Assistive Technologies*. ACM, 2002.

[49] Lewis R. Karl, Michael Pettey, and Ben Shneiderman. Speech versus mouse commands for word processing: An empirical evaluation. *International Journal of Man-Machine Studies*, 39(4):667–687, 1993.

[50] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, March 1993.

[51] Michael A. Klug. Visicola, a model and a language for visibility control in programming languages. *SIGPLAN Notices*, 26(2):51–63, 1991.

[52] Michael A. Klug. Basic operations of the visicola scope model. *SIGPLAN Notices*, 29(9):44–50, 1994.

[53] Michael A. Klug. Towards a classification of visibility rules. *SIGPLAN Notices*, 31(1):44–50, 1996.

[54] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th International Conference on Software Engineering*, pages 126–135, New York, NY, USA, 2005. ACM Press.

[55] Rainer Koppler. A systematic approach to fuzzy parsing. *Software—Practice and Experience*, 27(6):637–649, June 1997.

[56] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings on the 17th Annual International Conference on Computer Documentation*, pages 147–153. ACM Press, 1999.

[57] Michele Lanza and Stephane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 300–311, New York, NY, USA, 2001. ACM Press.

[58] Michael E. Lesk and Eric Schmidt. Lex — A lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[59] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. In *Proceedings of the Third ACM SIGSOFT '88 Symposium on Software Development Environments*, pages 173–182, November 1988. Published as SIGSOFT Software Engineering Notes, volume 13, number 5.

[60] Hugo Liu and Henry Lieberman. Metafor: Visualizing stories as code. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, pages 305–307, New York, NY, USA, 2005. ACM Press.

[61] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond AOP: toward naturalistic programming. *SIGPLAN Notices*, 38(12):34–43, 2003.

[62] I. Scott MacKenzie and William Buxton. Extending Fitt's law to two-dimensional tasks. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, pages 219–226, 1992.

[63] Bill Z. Manaris, Renée A. McCauley, and Valanne MacGyvers. An intelligent interface for keyboard and mouse control–providing full access to PC functionality via speech. In *FLAIRS Conference*, pages 182–188, 2001.

[64] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM Press.

[65] David Patrick Marin. What motivates programmers to comment? Master's thesis, EECS Department, University of California, Berkeley, November 2005.

[66] Gregory R. McArthur. If writers can't program and programmers can't write, who's writing user documentation? In *Proceedings of the Fourth International Conference on Systems Documentation*, pages 62–70. ACM Press, 1985.

[67] Scott McPeak. Elkhound: A fast, practical GLR parser generator. In *13th International Conference on Compiler Construction*, pages 73–88, 2004.

[68] Brad A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[69] Nuance Communications, Inc. Dragon NaturallySpeaking. http://www.nuance.com/naturallyspeaking/.

[70] U.S. Department of Labor. Bureau of labor statistics, injuries, illnesses and fatalities home page. http://stats.bls.gov/iif/home.htm.

[71] J. C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.

[72] Joseph C. H. Park. A new LALR formalism. *SIGPLAN Notices*, 17(7):47–61, 1982.

[73] Joseph C. H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, 1987.

[74] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[75] Vern Paxson. Flex – fast lexical analyzer generator. Free Software Foundation, 1988.

[76] John Plevyak. D Parser Homepage. http://dparser.sourceforge.net.

[77] Blaine A. Price, Ronald Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[78] David Price, Dana Dahlstrom, Ben Newton, and Joseph Zachary. Off to see the wizard: Using a "Wizard of Oz" study to learn how to design a spoken language interface for programming. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference*, pages T2G23–29, November 2002.

[79] David Price, Ellen Rillof, Joseph Zachary, and Brandon Harvey. NaturalJava: A natural language interface for programming in Java. In *Proceedings of the International Conference on Intelligent User Interfaces*, Short Paper/Poster/Demonstration, pages 207–211, 2000.

[80] T. V. Raman. Emacspeak – direct speech access. In *Proceedings of the First Annual ACM Conference on Assistive Technologies*, pages 32–36, 1996.

[81] RealVNC. VNC: Virtual Network Computing. http://www.realvnc.com/.

[82] S. P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, 1995.

[83] Steven P. Reiss. Generation of compiler symbol processing mechanisms from specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):127–163, April 1983.

[84] Jan Rekers. *Parser Generation for Interactive Environments*. Ph.D. dissertation, University of Amsterdam, 1992.

[85] R. Douglas Riecken, Jurgen Koenemann-Belliveau, and Scott P. Robertson. What do expert programmers communicate by means of descriptive commenting? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 177–195, 1991.

[86] David Roach, Hal Berghel, and John R. Talburt. An interactive source commenter for Prolog programs. In *Proceedings of the 8th Annual Conference on Systems Documentation*, pages 141–145. ACM Press, 1990.

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

[88] D. J. Salomon and G. V. Cormack. Corrections to the paper: Scannerless NSLR(1) Parsing of Programming Languages. *ACM SIGPLAN Notices*, 24(11):80–83, November 1989.

[89] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 170–178, 1989.

[90] Shairaj Shaik, Raymond Corvin, Rajesh Sudarsan, Faizan Javed, Qasim Ijaz, Suman Roychoudhury, Jeff Gray, and Barrett R. Bryant. SpeechClipse: an Eclipse speech plug-in. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, pages 84–88. ACM Press, 2003.

[91] Ann C. Smith, Joan M. Francioni, and Sam D. Matzek. A Java programming tool for students with visual disabilities. In *Fourth Annual ACM Conference on Assistive Technologies*, pages 142–148. ACM, 2000.

[92] Lindsey Snell. An investigation into programming by voice and development of a toolkit for writing voice-controlled applications. M.Eng. Report, Imperial College of Science, Technology and Medicine, London, June 2000.

[93] Sun Microsystems Inc. Java Speech API. http://java.sun.com/products/java-media/speech/index.jsp.

[94] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 266–278. IBM Press, 2003.

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

[96] Vicente Lopez Trueba, Julio Cesar Leon Carrillo, Oscar Olvera Posadas, and Carlos Ortega Hurtado. A system for automatic Cobol program documentation. In *Third International Conference on Systems Documentation*, pages 36–43, 1984.

[97] Michael L. Van De Vanter and Marat Boshernitsan. Displaying and editing source code in software engineering environments. In *Proceedings of Second International Symposium on Constructing Software Engineering Tools*, pages 39–48, Limerick, Ireland, 2000.

[98] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *12th International Conference on Compiler Construction*, pages 143–158, 2002. In Lecture Notes in Computer Science 2304.

[99] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[100] Eelco Visser. *Syntax Definition for Language Prototyping*. Ph.d. dissertation, University of Amsterdam, 1997.

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

[102] Scott A. Vorthmann. *Syntax-directed editor support for incremental consistency maintenance*. Ph.D. dissertation, Georgia Institute of Technology, 1990.

[103] Scott A. Vorthmann. Modelling and specifying name visibility and binding semantics. Technical Report CMU//CS-93-158, Carnegie-Mellon University, December 06 1993.

[104] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. Ph.D. dissertation, University of California, Berkeley, March 11, 1998. Technical Report UCB/CSD-97-946.

[105] Tim A. Wagner and Susan L. Graham. Efficient self-versioning documents. In *Proceedings of 42nd IEEE International Computer Conference*, San Jose, CA, 1997.

[106] Tim A. Wagner and Susan L. Graham. General incremental lexical analysis, 1997. Unpublished.

[107] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, 1997.

[108] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, September 1998.

[109] XEmacs: The next generation of Emacs. http://www.xemacs.org.

[110] Katherine Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10:825–836, 1998.

[111] J. M. Yohe. An overview of programming practices. *ACM Computing Surveys*, 6(4):221–245, 1974.

Appendix A

Java Code For Spoken Programs Study

This appendix contains the text of the one page Java program that participants in the Spoken Programs study (described in Chapter 2) read out loud.

```
package yoyo;

import java.util.Stack;

public class Pool implements Runnable {

  static Object undefined = new Object();

  static Object[][][] pool;
  static int[] ptrs;

  static {
    pool = new Object[8][10][];
    ptrs = new int[8];
    for(int i = 0; i<8; i++) {
      ptrs[i] = 0;
    }
    new Thread(new Pool()).start();
  }

  public static synchronized Object[] getArray(int size,
                                               Context c) {
    //System.out.println("get Array size: " + size);
    if (size >= 8) return new Object[size];
    //System.out.println("thepool: " +
    //                   YoYo.printToString(pool[size]));
    //System.out.println("ptrs[size]: " + ptrs[size]);
```

```
  Object[][] thepool = pool[size];
  if (ptrs[size] == 0) {
    Object[] output = new Object[size];
    for(int i = 0; i < size; i++) {
      output[i] = undefined;
    }
    return output;
  }
  return thepool[--ptrs[size]];
}

public static synchronized void dropArray(Object[] array,
                                          Context c) {
  int size = array.length;
  if (size >= 8) return;
  for(int i = 0; i<size; i++) {
    array[i] = undefined;
  }

  Object[][] thepool = pool[size];

  if (ptrs[size] >= thepool.length) {
    Object[][] newarray = new Object[thepool.length * 2][];
    System.arraycopy(thepool, 0, newarray, 0, thepool.length);
    pool[size] = newarray;
    thepool = newarray;
  }
  thepool[ptrs[size]++] = array;
}

// Cleanup thread. Runs every 5 seconds or so to clean
// out the pools.
public void run() {
  Thread thread = Thread.currentThread();
  while (true) {
    try {
      thread.sleep(5000);
    }
    catch (InterruptedException e) {}
    cleanPool();
  }
}

public static synchronized void cleanPool() {
```

```
    for(int i = 0; i < 8; i++) {
      Object[][] thepool = pool[i];
      int numentries = ptrs[i];
      if (numentries > 10) {
        ptrs[i] = 10;
        for(int j = 10; j < numentries; j++) {
          thepool[j] = null;
        }
      }
    }
  }
}
```

Appendix B

Spoken Java Language Specification

B.1 Lexical Specification

The following is the Whisk lexical specification for the Spoken Java language. First, each token is declared with its default spelling. Then a set of regular expression macro definitions are shown. Finally, all of the rules are given that map a regular expression to a Spoken Java token.

```
%token WSPC { spelling " " }
       LINE_COMMENT { spelling "comment" }
       BLOCK_COMMENT { spelling "block comment" }
       DOC_COMMENT { spelling "javadoc comment" }
       VOICECOMMENT

/* literals */

%token LongIntLiteral
       IntLiteral
       FloatLiteral
       DoubleLiteral
       CharacterLiteral
       StringLiteral

/* reserved words */

%token  ASSERT { spelling "assert" }
        BREAK { spelling "break" }
        CASE { spelling "case" }
        CATCH { spelling "catch" }
        CLASS { spelling "class" }
```

```
        CONTINUE { spelling "continue" }
        DEFAULT { spelling "default" }
        DO { spelling "do" }
        ELSE { spelling "else" }
        EXTENDS { spelling "extends"}
        FINALLY { spelling "finally" }
        FOR { spelling "for" }
        IF { spelling "if" }
        IMPLEMENTS { spelling "implements" }
        IMPORT { spelling "import" }
        INTERFACE { spelling "interface" }
        NEW { spelling "new" }
        PACKAGE { spelling "package" }
        RETURN { spelling "return" }
        STAR { alias '*' spelling "star" }
        SWITCH { spelling "switch" }
        THROW { spelling "throw" }
        THROWS { spelling "throws" }
        TRY { spelling "try" }
        WHILE { spelling "while" }

/* modifiers */

%token  ABSTRACT { spelling "abstract" }
        NATIVE { spelling "native" }
        PRIVATE { spelling "private" }
        PROTECTED { spelling "protected" }
        PUBLIC { spelling "public" }
        SYNCHRONIZED { spelling "synchronized" }
        STATIC { spelling "static" }
        FINAL { spelling "final" }
        TRANSIENT { spelling "transient" }
        STRICTFP { spelling "strict f p" }
        JVOLATILE { spelling "volatile" }

/* primitive types */

%token  JBOOLEAN { spelling "boolean" }
        JBYTE { spelling "byte" }
        JCHAR { spelling "character" }
        JFLOAT { spelling "float" }
        JINT { spelling "integer" }
        JLONG { spelling "long" }
        JDOUBLE { spelling "double" }
```

```
        JSHORT { spelling "short" }
        VOID { spelling "void" }

/* constants */

%token  FALSE_TOKEN { spelling "false" }
        TRUE_TOKEN { spelling "true" }
        NULL_TOKEN { spelling "null" }
        THIS { spelling "this" }
        SUPER { spelling "super" }

/* identifiers */

%token  IDENTIFIER { spelling "identifier" multitext
                     affects-ig }

/* separators */

%token  LBRACE { alias '{' spelling "{"  }
        RBRACE { alias '}' spelling "}" }
        LBRACKET { alias '[' spelling "[" }
        RBRACKET { alias ']' spelling "]" }
        SEMICOLON { alias ';' spelling ";" }
        COMMA { alias ',' spelling "," }
        DOT { alias '.' spelling "." }


/* operators */

%token  ASSIGN { alias '=' spelling "gets" }
        PLUS_ASSIGN { alias '+=' spelling "+=" }
        MINUS_ASSIGN { alias '-=' spelling "-=" }
        TIMES_ASSIGN { alias '*=' spelling "*=" }
        DIV_ASSIGN { alias '/=' spelling "/=" }
        AND_ASSIGN { alias '&=' spelling "&=" }
        XOR_ASSIGN { alias '^=' spelling "^=" }
        OR_ASSIGN { alias '|=' spelling "|=" }
        REM_ASSIGN { alias '%=' spelling "%=" }
        LSHIFT_ASSIGN { alias '<<=' spelling "<<=" }
        RARITHSHIFT_ASSIGN { alias '>>>=' spelling ">>>=" }
        RSHIFT_ASSIGN { alias '>>=' spelling ">>=" }
        HOOK { alias '?' spelling "?" }
        COLON { alias ':' spelling ":" }
        COND_OR { spelling "or" alias '||' }
```

```
        COND_AND { spelling "and" alias '&&' }
        OR { alias '|' spelling "|" }
        XOR { alias '^' spelling "^" }
        JAND { alias '&' spelling "&" }
        EQ { spelling "is equal to" alias '==' }
        NE { spelling "is not equal to" alias '!=' }
        LE { spelling "is less than or equal to" alias '<=' }
        GE { spelling "is greater than or equal to" alias '>=' }
        GT { alias '>' spelling "is greater than" }
        LT { alias '<' spelling "is less than" }
        INSTANCEOF { spelling "is an instance of" }
        RARITHSHIFT { spelling ">>>" alias '>>>' }
        RSHIFT { spelling ">>" alias '>>' }
        LSHIFT { spelling "<<" alias '<<' }
        PLUS { alias '+' spelling "plus" }
        MINUS { alias '-' spelling "minus" }
        TIMES { alias 'x' spelling "times" }
        DIV { alias '/' spelling "divided by" }
        REM { alias '%' spelling "mod" }
        JCAST { spelling "cast" }
        UPLUS { spelling "positive" }
        UMINUS { spelling "negative" }
        COND_NOT { alias '!' spelling "not" }
        NOT { alias '~' spelling "negate" }
        INCR { spelling "plus plus" alias '++'  }
        DECR { spelling "minus minus" alias '--' }
        LPAREN { alias '(' spelling "left paren" }
        RPAREN { alias ')' spelling "right paren" }

/* additional Spoken Java keywords */

%token  SETVAR { spelling "set" }
        BODY { spelling "body" }
        OFARRAY { spelling "of array"}
        SUB { spelling "sub" }
        TO { spelling "to" }
        THEN { spelling "then" }
        NOARGS { spelling "with no arguments" }
        OFSIZE { spelling "of size" }
        EMPTY { spelling "empty" }
        ARRAY { spelling "array" }
        ELEMENT { spelling "element" }
        THE { spelling "the" }
        A { spelling "a" }
```

```
/* additional Spoken Java punctuation equivalents */

%token  CLOSE_CLASS { spelling "close class" }
        CLOSE_IF { spelling "close if"}
        CLOSE_METHOD { spelling "close method" }
        CLOSE_FOR { spelling "close for" }
        CLOSE_WHILE { spelling "close while" }
        CLOSE_DO { spelling "close do" }
        CLOSE_FOREVER { spelling "close forever" }
        CLOSE_INTERFACE { spelling "close interface" }
        CLOSE_CONSTRUCTOR { spelling "close constructor" }


/* Lexical states */

%x IN_DOC_COMMENT IN_BLOCK_COMMENT


/* Lexical macros */

IDENTIFIER          <<[A-Za-z_$][A-Za-z_$0-9]*>>
DIGIT               <<[0-9]>>
HEXDIGIT            <<[A-Fa-f0-9]>>
OCTDIGIT            <<[0-7]>>
DECNUMBER           <<0|[1-9]{DIGIT}*>>
HEXNUMBER           <<0[Xx]{HEXDIGIT}+>>
OCTNUMBER           <<0{OCTDIGIT}+>>
DECLONG             <<{DECNUMBER}[Ll]>>
HEXLONG             <<{HEXNUMBER}[Ll]>>
OCTLONG             <<{OCTNUMBER}[Ll]>>
EXPONENT            <<[Ee][+-]?{DIGIT}+>>
FLOATBASE           <<((({DIGIT}+\.{DIGIT}*)|
                      ({DIGIT}*\.{DIGIT}+))
                      {EXPONENT}?)|({DIGIT}+{EXPONENT})>>

/* The {DIGIT}+ part of both of these is contrary
 * to the written spec, but the compiler accepts 0f
 * as a valid float literal, so I needed to add this
 * for compatibility.  */
DOUBLE              <<({FLOATBASE}[Dd]?)|({DIGIT}+[Dd])>>
FLOAT               <<({FLOATBASE}|({DIGIT}+))[Ff]>>

/* Harmonia is not Unicode clean and never will be.
```

```
 * For this reason, we accept unicode escapes
 * (and assume they are legal string characters) in
 * character/string constants, but do not allow them
 * in general */
OCTESCAPE           <<\\([0123]{OCTDIGIT}{2}|{OCTDIGIT}{1,2})>>
ONECHAR             <<[^\\\n\r"']|(\\[ntbrf\\'"])|
                      {OCTESCAPE}|(\\u{HEXDIGIT}{4})>>
CHARLITCHAR         <<{ONECHAR}|\">>
CHARACTER           <<"'"{CHARLITCHAR}"'">>
STRINGCHAR          <<{ONECHAR}|'>>
STRING              <<\"{STRINGCHAR}*\">>
WHITESPACE          <<[ \f\n\r\t\v]+>>

%%

/* Lexical rules */

/* Comments and whitespace */

<IN_DOC_COMMENT><<"*/">>    {= BEGIN(spoken_java_INITIAL);
                               RETURN_TOKEN(DOC_COMMENT); =}
<IN_DOC_COMMENT><<"close comment">>
                            {= BEGIN(spoken_java_INITIAL);
                               RETURN_TOKEN(DOC_COMMENT); =}
<IN_BLOCK_COMMENT><<"*/">>  {= BEGIN(spoken_java_INITIAL);
                               RETURN_TOKEN(BLOCK_COMMENT); =}
<IN_BLOCK_COMMENT><<"close comment">>
                            {= BEGIN(spoken_java_INITIAL);
                               RETURN_TOKEN(BLOCK_COMMENT); =}
<IN_DOC_COMMENT><<.|[\n\r]>>          {= yymore(); break; =}
<IN_BLOCK_COMMENT><<.|[\n\r]>>        {= yymore(); break; =}
<<"/*">>                {= BEGIN(spoken_java_IN_BLOCK_COMMENT);
                           yymore(); break; =}
<<"block comment">>     {= BEGIN(spoken_java_IN_BLOCK_COMMENT);
                           yymore(); break; =}
<<"/**">>               {= BEGIN(spoken_java_IN_DOC_COMMENT);
                           yymore(); break; =}
<<"javadoc comment">>   {= BEGIN(spoken_java_IN_DOC_COMMENT);
                           yymore(); break; =}
<<"/**/">>              {= RETURN_TOKEN(BLOCK_COMMENT); =}
<<"empty comment">>     {= RETURN_TOKEN(BLOCK_COMMENT); =}
<<"//"[^\n\r]*>>        {= RETURN_TOKEN(LINE_COMMENT); =}
<<"comment"[^\n\r]*>>   {= RETURN_TOKEN(LINE_COMMENT); =}
```

```
<<{WHITESPACE}>>          {= RETURN_TOKEN(WSPC); =}

   /* literals */

<<{DECLONG}|{HEXLONG}|{OCTLONG}>>
                          {= RETURN_TOKEN(LongIntLiteral); =}
<<{DECNUMBER}|{HEXNUMBER}|{OCTNUMBER}>>
                          {= RETURN_TOKEN(IntLiteral); =}
<<{CHARACTER}>>           {= RETURN_TOKEN(CharacterLiteral); =}
<<{FLOAT}>>               {= RETURN_TOKEN(FloatLiteral); =}
<<{DOUBLE}>>              {= RETURN_TOKEN(DoubleLiteral); =}
<<{STRING}>>              {= RETURN_TOKEN(StringLiteral); =}

   /* reserved words */

<<"to">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(TO, IDENTIFIER); =}
<<"abstract">>       {= RETURN_TOKEN(ABSTRACT); =}
<<"assert">>         {= RETURN_TOKEN(ASSERT); =}
<<"boolean">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JBOOLEAN, IDENTIFIER); =}
<<"break">>          {= RETURN_TOKEN(BREAK); =}
<<"byte">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JBYTE, IDENTIFIER); =}
<<"case">>           {= RETURN_TOKEN(CASE); =}
<<"catch">>          {= RETURN_TOKEN(CATCH); =}
<<"char">>           {= RETURN_TOKEN(JCHAR); =}
<<"class">>          {= RETURN_TOKEN(CLASS); =}
<<"continue">>       {= RETURN_TOKEN(CONTINUE); =}
<<"default">>        {= RETURN_TOKEN(DEFAULT); =}
<<"do">>             {= RETURN_TOKEN(DO); =}
<<"double">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JDOUBLE, IDENTIFIER); =}
<<"else">>           {= RETURN_TOKEN(ELSE); =}
<<"then">>           {= RETURN_TOKEN(THEN); =}
<<"extends">>        {= RETURN_TOKEN(EXTENDS); =}
<<"false">>          {= RETURN_TOKEN(FALSE_TOKEN); =}
<<"final">>          {= RETURN_TOKEN(FINAL); =}
<<"finally">>        {= RETURN_TOKEN(FINALLY); =}
<<"float">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JFLOAT, IDENTIFIER); =}
<<"for">>            {= RETURN_TOKEN(FOR); =}
<<"if">>             {= RETURN_TOKEN(IF); =}
```

```
<<"implements">>    {= RETURN_TOKEN(IMPLEMENTS ); =}
<<"import">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(IMPORT, IDENTIFIER);=}
<<"instanceof">>    {= RETURN_TOKEN(INSTANCEOF); =}
<<"int">>           {= RETURN_TOKEN(JINT); =}
<<"interface">>     {= RETURN_TOKEN(INTERFACE); =}
<<"long">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JLONG, IDENTIFIER); =}
<<"native">>        {= RETURN_TOKEN(NATIVE); =}
<<"new">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(NEW, IDENTIFIER); =}
<<"null">>          {= RETURN_TOKEN(NULL_TOKEN); =}
<<"package">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(PACKAGE, IDENTIFIER); =}
<<"private">>       {= RETURN_TOKEN(PRIVATE); =}
<<"protected">>     {= RETURN_TOKEN(PROTECTED); =}
<<"public">>        {= RETURN_TOKEN(PUBLIC); =}
<<"return">>        {= RETURN_TOKEN(RETURN); =}
<<"short">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JSHORT, IDENTIFIER); =}
<<"static">>        {= RETURN_TOKEN(STATIC); =}
<<"super">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(SUPER, IDENTIFIER); =}
<<"switch">>        {= RETURN_TOKEN(SWITCH); =}
<<"synchronized">>  {= RETURN_TOKEN(SYNCHRONIZED); =}
<<"this">>          {= RETURN_TOKEN(THIS); =}
<<"throw">>         {= RETURN_TOKEN(THROW); =}
<<"throws">>        {= RETURN_TOKEN(THROWS); =}
<<"transient">>     {= RETURN_TOKEN(TRANSIENT); =}
<<"true">>          {= RETURN_TOKEN(TRUE_TOKEN); =}
<<"try">>           {= RETURN_TOKEN(TRY); =}
<<"void">>          {= RETURN_TOKEN(VOID); =}
<<"volatile">>      {= RETURN_TOKEN(JVOLATILE); =}
<<"while">>         {= RETURN_TOKEN(WHILE); =}

<<"no arguments">>  {= RETURN_TOKEN(NOARGS); =}
<<"with no arguments">>   {= RETURN_TOKEN(NOARGS); =}
<<"takes no arguments">>  {= RETURN_TOKEN(NOARGS); =}
<<"body">>          {= RETURN_TOKEN(BODY); =}
<<"start body">>    {= RETURN_TOKEN(BODY); =}
<<"begin body">>    {= RETURN_TOKEN(BODY); =}

<<"self">>          {= RETURN_TOKEN(THIS); =}
```

```
<<"set variable">>  {= RETURN_TOKEN(SETVAR); =}
<<"set">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(SETVAR, IDENTIFIER); =}
<<"make">>          {= RETURN_TOKEN(SETVAR); =}
<<"initialize">>    {= RETURN_TOKEN(SETVAR); =}

<<"cast">>          {= RETURN_TOKEN(JCAST); =}
<<"typecast">>      {= RETURN_TOKEN(JCAST); =}

<<"strict f p">>    {= RETURN_TOKEN(STRICTFP); =}

<<"bool">>          {= RETURN_TOKEN(JBOOLEAN); =}
<<"integer">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JINT, IDENTIFIER); =}

<<"char">>          {= RETURN_TOKEN(JCHAR); =}
<<"character">>     {= RETURN_TOKEN(JCHAR); =}
<<"car">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(JCHAR, IDENTIFIER); =}

<<"empty">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(EMPTY, IDENTIFIER); =}

<<"dot">>           {= RETURN_TOKEN(DOT); =}
<<"point">>         {= RETURN_TOKEN(DOT); =}

<<"semicolon">>     {= RETURN_TOKEN(SEMICOLON); =}

<<"comma">>         {= RETURN_TOKEN(COMMA); =}
<<"array">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(ARRAY, IDENTIFIER); =}
<<"of array">>      {= RETURN_TOKEN(OFARRAY); =}
<<"of size">>       {= RETURN_TOKEN(OFSIZE); =}
<<"element">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(ELEMENT, IDENTIFIER); =}

<<"close class">>   {= RETURN_TOKEN(CLOSE_CLASS); =}
<<"end class">>     {= RETURN_TOKEN(CLOSE_CLASS); =}

<<"close interface">>  {= RETURN_TOKEN(CLOSE_INTERFACE); =}
<<"end interface">>    {= RETURN_TOKEN(CLOSE_INTERFACE); =}

<<"close if">>      {= RETURN_TOKEN(CLOSE_IF); =}
<<"end if">>        {= RETURN_TOKEN(CLOSE_IF); =}
```

```
<<"close method">>  {= RETURN_TOKEN(CLOSE_METHOD); =}
<<"end method">>    {= RETURN_TOKEN(CLOSE_METHOD); =}

<<"close constructor">>  {= RETURN_TOKEN(CLOSE_CONSTRUCTOR); =}
<<"end constructor">>    {= RETURN_TOKEN(CLOSE_CONSTRUCTOR); =}

<<"close for">>     {= RETURN_TOKEN(CLOSE_FOR); =}
<<"end for">>       {= RETURN_TOKEN(CLOSE_FOR); =}

<<"close while">>   {= RETURN_TOKEN(CLOSE_WHILE); =}
<<"end while">>     {= RETURN_TOKEN(CLOSE_WHILE); =}

<<"close do">>      {= RETURN_TOKEN(CLOSE_DO); =}
<<"end do">>        {= RETURN_TOKEN(CLOSE_DO); =}

<<"close forever">> {= RETURN_TOKEN(CLOSE_FOREVER); =}
<<"end forever">>   {= RETURN_TOKEN(CLOSE_FOREVER); =}

<<"equals equals">> {= RETURN_TOKEN(EQ); =}
<<"equal equal">>   {= RETURN_TOKEN(EQ); =}
<<"equals">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(EQ, ASSIGN); =}
<<"is equal to">>   {= RETURN_TOKEN(EQ); =}
<<"equal to">>      {= RETURN_TOKEN(EQ); =}

<<"plus equals">>   {= RETURN_TOKEN(PLUS_ASSIGN); =}
<<"minus equals">>  {= RETURN_TOKEN(MINUS_ASSIGN); =}

<<"multiply equals">>   {= RETURN_TOKEN(TIMES_ASSIGN); =}
<<"times equals">>  {= RETURN_TOKEN(TIMES_ASSIGN); =}
<<"star equals">>   {= RETURN_TOKEN(TIMES_ASSIGN); =}

<<"divide equals">> {= RETURN_TOKEN(DIV_ASSIGN); =}
<<"div equals">>    {= RETURN_TOKEN(DIV_ASSIGN); =}

<<"and equals">>    {= RETURN_TOKEN(AND_ASSIGN); =}

<<"x or equals">>   {= RETURN_TOKEN(XOR_ASSIGN); =}
<<"xor equals">>    {= RETURN_TOKEN(XOR_ASSIGN); =}

<<"or equals">>     {= RETURN_TOKEN(OR_ASSIGN); =}
<<"mod equals">>    {= RETURN_TOKEN(REM_ASSIGN); =}
<<"left shift equals">>
```

```
                    {= RETURN_TOKEN(LSHIFT_ASSIGN); =}
<<"l s h equals">>  {= RETURN_TOKEN(LSHIFT_ASSIGN); =}

<<"right shift equals">>
                    {= RETURN_TOKEN(RSHIFT_ASSIGN); =}
<<"r s h equals">>  {= RETURN_TOKEN(RSHIFT_ASSIGN); =}

<<"arithmetic right shift equals">>
                    {= RETURN_TOKEN(RARITHSHIFT_ASSIGN); =}
<<"a r s h equals">>
                    {= RETURN_TOKEN(RARITHSHIFT_ASSIGN); =}

<<"or">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(COND_OR, OR); =}
<<"and">>
     {= RETURN_TOKEN_WITH_3_ALTERNATES(COND_AND, COMMA, JAND); =}
<<"bitwise or">>    {= RETURN_TOKEN(OR); =}

<<"x or">>          {= RETURN_TOKEN(XOR); =}
<<"xor">>           {= RETURN_TOKEN(XOR); =}

<<"bitwise and">>   {= RETURN_TOKEN(JAND); =}

<<"gets">>          {= RETURN_TOKEN(ASSIGN); =}

<<"not equal">>     {= RETURN_TOKEN(NE); =}
<<"not equals">>    {= RETURN_TOKEN(NE); =}
<<"does not equal">>  {= RETURN_TOKEN(NE); =}
<<"is not equal to">> {= RETURN_TOKEN(NE); =}
<<"not equal to">>    {= RETURN_TOKEN(NE); =}

<<"less than or equal to">>      {= RETURN_TOKEN(LE); =}
<<"is less than or equal to">>   {= RETURN_TOKEN(LE); =}
<<"greater than or equal to">>   {= RETURN_TOKEN(GE); =}
<<"is greater than or equal to">>   {= RETURN_TOKEN(GE); =}
<<"less than">>         {= RETURN_TOKEN(LT); =}
<<"is less than">>      {= RETURN_TOKEN(LT); =}
<<"greater than">>      {= RETURN_TOKEN(GT); =}
<<"is greater than">>   {= RETURN_TOKEN(GT); =}
<<"instance of">>       {= RETURN_TOKEN(INSTANCEOF); =}
<<"is an instance of">> {= RETURN_TOKEN(INSTANCEOF); =}

<<"left shift">>     {= RETURN_TOKEN(LSHIFT); =}
<<"l s h">>          {= RETURN_TOKEN(LSHIFT); =}
```

```
<<"right shift">>    {= RETURN_TOKEN(RSHIFT); =}
<<"r s h">>          {= RETURN_TOKEN(RSHIFT); =}

<<"arithmetic right shift">>
                     {= RETURN_TOKEN(RARITHSHIFT); =}
<<"a r s h">>        {= RETURN_TOKEN(RARITHSHIFT); =}

<<"plus">>           {= RETURN_TOKEN(PLUS); =}
<<"add">>            {= RETURN_TOKEN(PLUS); =}

<<"minus">>          {= RETURN_TOKEN(MINUS); =}
<<"subtract">>       {= RETURN_TOKEN(MINUS); =}

<<"positive">>       {= RETURN_TOKEN(UPLUS); =}
<<"negative">>       {= RETURN_TOKEN(UMINUS); =}

<<"times">>          {= RETURN_TOKEN(TIMES); =}
<<"star">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(STAR, TIMES); =}

<<"divided by">>     {= RETURN_TOKEN(DIV); =}
<<"div">>            {= RETURN_TOKEN(DIV); =}

<<"mod">>            {= RETURN_TOKEN(REM); =}

<<"not">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(COND_NOT, NOT); =}
<<"bang">>           {= RETURN_TOKEN(COND_NOT); =}

<<"negate">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(NOT, IDENTIFIER); =}
<<"invert">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(NOT, IDENTIFIER); =}

<<"plus plus">>      {= RETURN_TOKEN(INCR); =}
<<"increment">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(INCR, IDENTIFIER); =}
<<"minus minus">>    {= RETURN_TOKEN(DECR); =}
<<"decrement">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(DECR, IDENTIFIER); =}

/* punctuation */
```

```
<<"left paren">>     {= RETURN_TOKEN(LPAREN); =}
<<"right paren">>    {= RETURN_TOKEN(RPAREN); =}
<<"open paren">>     {= RETURN_TOKEN(LPAREN); =}
<<"close paren">>    {= RETURN_TOKEN(RPAREN); =}

<<"left bracket">>   {= RETURN_TOKEN(LBRACKET); =}
<<"right bracket">>  {= RETURN_TOKEN(RBRACKET); =}
<<"open bracket">>   {= RETURN_TOKEN(LBRACKET); =}
<<"close bracket">>  {= RETURN_TOKEN(RBRACKET); =}
<<"sub">>            {= RETURN_TOKEN(SUB); =}

<<"left square bracket">>    {= RETURN_TOKEN(LBRACKET); =}
<<"right square bracket">>   {= RETURN_TOKEN(RBRACKET); =}
<<"open square bracket">>    {= RETURN_TOKEN(LBRACKET); =}
<<"close square bracket">>   {= RETURN_TOKEN(RBRACKET); =}

<<"left brace">>     {= RETURN_TOKEN(LBRACE); =}
<<"right brace">>    {= RETURN_TOKEN(RBRACE); =}
<<"open brace">>     {= RETURN_TOKEN(LBRACE); =}
<<"close brace">>    {= RETURN_TOKEN(RBRACE); =}

<<"left curly brace">>       {= RETURN_TOKEN(LBRACE); =}
<<"right curly brace">>      {= RETURN_TOKEN(RBRACE); =}
<<"open curly brace">>       {= RETURN_TOKEN(LBRACE); =}
<<"close curly brace">>      {= RETURN_TOKEN(RBRACE); =}

<<"the">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(THE, IDENTIFIER); =}
<<"a">>
     {= RETURN_TOKEN_WITH_2_ALTERNATES(A, IDENTIFIER); =}


   /* identifiers */

<<{IDENTIFIER}>>     {= RETURN_TOKEN(IDENTIFIER); =}

   /* separators */

<<"{">>             {= RETURN_TOKEN(LBRACE); =}
<<"}">>             {= RETURN_TOKEN(RBRACE); =}
<<"(">>             {= RETURN_TOKEN(LPAREN); =}
<<")">>             {= RETURN_TOKEN(RPAREN); =}
<<"[">>             {= RETURN_TOKEN(LBRACKET); =}
<<"]">>             {= RETURN_TOKEN(RBRACKET); =}
```

```
<<";">>             {= RETURN_TOKEN(SEMICOLON); =}
<<",">>             {= RETURN_TOKEN(COMMA); =}
<<".">>             {= RETURN_TOKEN(DOT); =}

   /* operators */

<<"=">>             {= RETURN_TOKEN(ASSIGN); =}
<<">">>             {= RETURN_TOKEN(GT); =}
<<"<">>             {= RETURN_TOKEN(LT); =}
<<"!">>             {= RETURN_TOKEN(COND_NOT); =}
<<"~">>             {= RETURN_TOKEN(NOT); =}
<<"?">>             {= RETURN_TOKEN(HOOK); =}
<<":">>             {= RETURN_TOKEN(COLON); =}
<<"==">>            {= RETURN_TOKEN(EQ); =}
<<"<=">>            {= RETURN_TOKEN(LE); =}
<<">=">>            {= RETURN_TOKEN(GE); =}
<<"!=">>            {= RETURN_TOKEN(NE); =}
<<"&&">>            {= RETURN_TOKEN(COND_AND); =}
<<"||">>            {= RETURN_TOKEN(COND_OR); =}
<<"++">>            {= RETURN_TOKEN(INCR); =}
<<"--">>            {= RETURN_TOKEN(DECR); =}
<<"+">>             {= RETURN_TOKEN(PLUS); =}
<<"-">>             {= RETURN_TOKEN(MINUS); =}
<<"*">>
   {= RETURN_TOKEN_WITH_2_ALTERNATES(STAR, TIMES); =}
<<"/">>             {= RETURN_TOKEN(DIV); =}
<<"&">>             {= RETURN_TOKEN(JAND); =}
<<"|">>             {= RETURN_TOKEN(OR); =}
<<"^">>             {= RETURN_TOKEN(XOR); =}
<<"%">>             {= RETURN_TOKEN(REM); =}
<<">>">>            {= RETURN_TOKEN(RARITHSHIFT); =}
<<">>>">>           {= RETURN_TOKEN(RSHIFT); =}
<<"<<">>            {= RETURN_TOKEN(LSHIFT); =}
<<"+=">>            {= RETURN_TOKEN(PLUS_ASSIGN); =}
<<"-=">>            {= RETURN_TOKEN(MINUS_ASSIGN); =}
<<"*=">>            {= RETURN_TOKEN(TIMES_ASSIGN); =}
<<"/=">>            {= RETURN_TOKEN(DIV_ASSIGN); =}
<<"&=">>            {= RETURN_TOKEN(AND_ASSIGN); =}
<<"^=">>            {= RETURN_TOKEN(XOR_ASSIGN); =}
<<"|=">>            {= RETURN_TOKEN(OR_ASSIGN); =}
<<"%=">>            {= RETURN_TOKEN(REM_ASSIGN); =}
<<"<<=">>           {= RETURN_TOKEN(LSHIFT_ASSIGN); =}
<<">>=">>           {= RETURN_TOKEN(RARITHSHIFT_ASSIGN); =}
<<">>>=">>          {= RETURN_TOKEN(RSHIFT_ASSIGN); =}
```

```
   /* lexical errors */

<*><<.>>            {= ERROR_ACTION; =}
```

B.2 Spoken Java Grammar

The following is a Ladle grammar for the Spoken Java grammar. It is based on the Java grammar used in the Harmonia program analysis tool (available in source code form on the Web [36]). Productions that are modified from the original grammar are noted, as are new productions.

```
%import-tokens spoken_java "spoken_java.wsk" default
%grammar-name spoken_java

%whitespace WSPC THE A
%comment    LINE_COMMENT BLOCK_COMMENT
            DOC_COMMENT VOICECOMMENT

/* operators in precedence order (lowest to highest) */

%right      ASSIGN PLUS_ASSIGN MINUS_ASSIGN TIMES_ASSIGN \
            DIV_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN \
            REM_ASSIGN LSHIFT_ASSIGN RARITHSHIFT_ASSIGN \
            RSHIFT_ASSIGN
%right      HOOK COLON
%left       COND_OR
%left       COND_AND
%left       OR
%left       XOR
%left       JAND
%left       EQ NE
%left       LE GE GT LT INSTANCEOF
%left       RARITHSHIFT RSHIFT LSHIFT
%left       PLUS MINUS
%left       STAR DIV REM
%nonassoc   JCAST !CAST
%right      UPLUS UMINUS COND_NOT NOT INCR DECR
%left       !POSTINCR !POSTDECR
%nonassoc   LPAREN RPAREN
%nonassoc   !LOWER_THAN_ELSE
%nonassoc   ELSE
```

```
%nonassoc   !LOWER_THAN_DOT
%nonassoc   DOT
%nonassoc   !LOWER_THAN_LBRACKET
%nonassoc   LBRACKET

%nonterm CompileUnit { can-isolate }
%nonterm PackageDecl { can-isolate }
%nonterm ImportDecl { can-isolate }
%nonterm TypeDecl { can-isolate }
%nonterm ClassBody { can-isolate }
%nonterm ClassBody2 { can-isolate }
%nonterm InterfaceBody { can-isolate }
%nonterm VarInitializer { can-isolate }
%nonterm MethodBody { can-isolate }

%%


CompileUnit
   : pDecl:(pDecl:PackageDecl)? iDecls:(iDecl:ImportDecl)*
      tDecls:(tDecl:TypeDecl)*
               { classname CompilationUnit }
   ;

/* semicolon removed */
PackageDecl
   : package_kw:PACKAGE name:Name
               { classname PackageDeclaration }
   ;

/* semicolon and dot removed */
ImportDecl
   : import_kw:IMPORT name:Name ondemand:('*')?
               { classname ImportDeclaration }
   ;

TypeDecl
   : cDecl:ClassDecl
               { classname ClassTypeDeclaration }
   | iDecl:InterfaceDecl
               { classname InterfaceTypeDeclaration }
   | semi:';'
               { classname SpuriousToplevelSemi }
   ;
```

```
Modifier
   : mod:PUBLIC         { classname PublicMod }
   | mod:PROTECTED      { classname ProtectedMod }
   | mod:PRIVATE        { classname PrivateMod }
   | mod:STATIC         { classname StaticMod }
   | mod:FINAL          { classname FinalMod }
   | mod:ABSTRACT       { classname AbstractMod }
   | mod:NATIVE         { classname NativeMod }
   | mod:SYNCHRONIZED   { classname SynchronizedMod }
   | mod:STRICTFP       { classname StrictFPMod }
   | mod:TRANSIENT      { classname TransientMod }
   | mod:JVOLATILE      { classname VolatileMod }
   ;

/* classes */

ClassDecl
   : mods:(mod:Modifier)* class_kw:CLASS name:Ident
      extends:(extends:ExtendsDecl)?
      implements:(implements:ImplementsDecl)? body:ClassBody
               { classname ClassDeclaration }
   ;

ExtendsDecl
   : extends_kw:EXTENDS name:Name
               { classname ClassExtends }
   ;

/* Comma separators removed from names list */
ImplementsDecl
   : implements_kw:IMPLEMENTS names:(name:Name)+
               { classname ClassImplements }
   ;

/* Class2 is new. Braces are removed, and optional terminator
 * added */
ClassBody
   : lbrace:'{' decls:(decl:ClassBodyDecl)* rbrace:'}'
               { classname Class }
   | decls:(decl:ClassBodyDecl)* CLOSE_CLASS?
               { classname Class2 }
   ;
```

```
/* ClassBody2 is new. Used for anonymous class definitions. */
/* Class4 is a plus list, Class2 is a star list. */
ClassBody2
   : lbrace:'{' decls:(decl:ClassBodyDecl)* rbrace:'}'
               { classname Class3 }
   | decls:(decl:ClassBodyDecl)+ CLOSE_CLASS?
               { classname Class4 }
   ;


ClassBodyDecl
   : decl:FieldDecl      { classname ClassFieldDecl }
   | decl:MethodDecl     { classname ClassMethodDecl }
   | decl:StaticInitDecl { classname ClassStaticInitDecl }
   | decl:InitDecl       { classname ClassInitDecl }
   | decl:ConstructDecl  { classname ClassConstructorDecl }
   | decl:ClassDecl      { classname ClassIClassDecl }
   | decl:InterfaceDecl  { classname ClassIInterfaceDecl }
   | semi:';'            { classname SpuriousClassSemi }
   ;

/* interfaces */

InterfaceDecl
   : mods:(mod:Modifier)* interface_kw:INTERFACE name:Ident
      extends:(extends:IntExtendsDecl)? body:InterfaceBody
               { classname InterfaceDeclaration }
   ;

/* Comma removed from names list */
IntExtendsDecl
   : extends_kw:EXTENDS names:(name:Name)+
               { classname InterfaceExtends }
   ;

/* Interface2 is new. Braces are removed, and optional
 * terminator added. Note that both close class and
 * close interface are allowed to close an interface. */
InterfaceBody
   : lbrace:'{' decls:(decl:InterfaceBodyDecl)* rbrace:'}'
               { classname Interface }
   | decls:(decl:InterfaceBodyDecl)*
       ( (CLOSE_INTERFACE | CLOSE_CLASS ) )?
               { classname Interface2 }
```

```
   ;

InterfaceBodyDecl
   : decl:ConstantDecl
               { classname InterfaceConstantDecl }
   | decl:AbstractMethodDecl
               { classname InterfaceAbstrMethodDecl }
   | decl:ClassDecl
               { classname InterfaceIClassDecl }
   | decl:InterfaceDecl
               { classname InterfaceIInterfaceDecl }
   | semi:';'
               { classname SpuriousInterfaceSemi }
   ;

/* Semicolon is optional */
ConstantDecl
   : mods:(mod:Modifier)* vDecl:VariableDecl semi:(semi:';')?
               { classname ConstantDeclaration }
   ;

/* Comma separators are removed from vDecls list */
VariableDecl
   : typ:Type vDecls:(vDecl:VarDeclarator)+
               { classname VariableDeclaration }
   ;

/* Semicolon is optional. */
AbstractMethodDecl
   : mods:(mod:Modifier)* result:ResultType
       declr:MethodDeclarator throws:(throws:Throws)?
       semi:(semi:';')?
               { classname AbstractMethodDeclaration }
   ;

/* fields */

/* Field keyword added. Semicolon is optional */
FieldDecl
   : mods:(mod:Modifier)* vDecl:VariableDecl
       semi:(semi:';')?
               { classname FieldDeclaration }
   ;
```

```
/* variable declarations */

VarDeclarator
   : id:Ident dims:(dim:Dim)* init:('=' init:VarInitializer)?
               { classname VariableDeclarator }
   ;

VarInitializer
   : expr:Expr           { classname ExprVarInitializer }
   | init:ArrayInit      { classname ArrayVarInitializer }
   ;

/* Comma removed as list separators. */
ArrayInit
   : lbrace:'{' inits:(init:VarInitializer)* rbrace:'}'
               { classname ArrayInitializer }
   ;

/* methods */

/* Method keyword added. */
MethodDecl
   : mods:(mod:Modifier)* result:ResultType
      declr:MethodDeclarator throws:(throws:Throws)?
      body:MethodBody
               { classname MethodDeclaration }
   ;

/* Optional body keyword added. Made optional with new open brace.
 *  Optional terminator added */
MethodBody
   : ( (BODY | '{') )? block:Block (CLOSE_METHOD)?
               { classname BlockMethodBody }
   | semi:';'
               { classname AbstractMethodBody }
   ;

/* Parens around argument list are removed. */
 * Added MethodSignature2. Used when method declaration has no
 * arguments */
MethodDeclarator
   : id:Ident params:(param:FormalParameter)*[',']
       dims:(dim:Dim)*
               { classname MethodSignature }
```

```
   | id:Ident NOARGS dims:(dim:Dim)*
               { classname MethodSignature2 }
   ;

FormalParameter
   : final:(FINAL)? type:Type id:Ident dims:(dim:Dim)*
               { classname FormalParam }
   ;

/* Comma separators removed from names list */
Throws   : throws_kw:THROWS names:(name:Name)+
               { classname ThrowsDecl }
   ;

/* initializers */

/* Body keyword introduced. Make optional with left brace. */
InitDecl
   : (BODY | '{') block:Block
               { classname Initializer }
   ;

/* Body keyword introduced. Make optional with left brace. */
StaticInitDecl
   : static_kw:STATIC (BODY | '{') block:Block
               { classname StaticInitializer }
   ;

/* constructors */

/* Parens surrounding argument
 * list are removed. ConstructorDeclaration2 is added. Used
 * when constructor declaration has no arguments. */
ConstructDecl
   : mods:(mod:Modifier)* id:Ident
       params:(param:FormalParameter)*[',']
       throws:(throws:Throws)? body:ConstructBody
               { classname ConstructorDeclaration }
   | mods:(mod:Modifier)* id:Ident
       NOARGS throws:(throws:Throws)? body:ConstructBody
               { classname ConstructorDeclaration2 }
   ;

/* Body keyword introduced. Make optional with left brace.
```

```
 * Optional terminator added after constructor body. Note
 * that terminator can use the word method or constructor. */
ConstructBody
   : ( (BODY | '{') )? explicit:(call:ExplicitConstructCall)?
       stmts:(stmt:BlockStmt)*
       ( ( CLOSE_METHOD | CLOSE_CONSTRUCTOR) )?
               { classname ConstructorBody }
   ;

/* Parens are removed from argument lists. Dot is optional.
 * Semicolon is removed. ThisConstructorCall2,
 * SuperConstructorCall2, and EnclSuperConstructorCall2
 * added to be used when constructor calls have no arguments */
ExplicitConstructCall
   : this_kw:THIS args:Args
               { classname ThisConstructorCall }
   | this_kw:THIS NOARGS
               { classname ThisConstructorCall2 }
   | super_kw:SUPER args:Args
               { classname SuperConstructorCall }
   | super_kw:SUPER NOARGS
               { classname SuperConstructorCall2 }
   | expr:NameOrPrimary '.'? super_kw:SUPER args:Args
               { classname EnclSuperConstructorCall }
   | expr:NameOrPrimary '.'? super_kw:SUPER NOARGS
               { classname EnclSuperConstructorCall2 }
   ;

/* blocks and statements */

/* Braces no longer surround stmts list */
Block   : stmts:(stmt:BlockStmt)*   { classname BlockBody }
   ;

/* Semicolon removed from BlockLocalVarDecl */
BlockStmt
   : decl:LocalVarDecl
               { classname BlockLocalVarDecl}
   | decl:ClassDecl
               { classname BlockInnerClassDecl }
   | decl:InterfaceDecl
               { classname BlockInnerInterfaceDecl }
   | stmt:Stmt
               { classname BlockStatement }
```

```
   ;

LocalVarDecl
   : final:(FINAL)? decl:VariableDecl
               { classname LocalVarDeclaration }
   ;

/* Body keyword added to BracedStatement. Made optional with
 * left brace.  IfThenStatement, IfThenElseStatement,
 * WhileStatement, and ForStatement have terminators.
 * IfThenStatement and IfThenElseStatement have added Then
 * keyword. Parens surrounding expr in IfThenStatement, expr
 * in IfTheElseStatement, expr in SwitchBlock, expr in
 * WhileStatement, expr in DoWhileStatement, init in
 * ForStatement, and expr in SynchronizedStatement are
 * removed. Recursive reference to BlockStmt in
 * LabelledStatement replaced by Block */
Stmt
   : semi:';'
               { classname EmptyStatement }
   | (BODY | '{') block:Block
               { classname BracedStatement }
   | label:Ident ':' stmt:Block
               { classname LabelledStatement }
   | expr:StatementExpr
               { classname ExpressionStatement }
   | if_kw:IF expr:Expr THEN stmt:Block CLOSE_IF
               { classname IfThenStatement
                 prec LOWER_THAN_ELSE }
   | if_kw:IF expr:Expr THEN tStmt:Block else_kw:ELSE
     fStmt:Block CLOSE_IF
               { classname IfThenElseStatement }
   | switch_kw:SWITCH expr:Expr block:SwitchBlock
               { classname SwitchStatement }
   | while_kw:WHILE expr:Expr DO stmt:Block CLOSE_WHILE
               { classname WhileStatement }
   | do_kw:DO stmt:Block while_kw:WHILE expr:Expr
               { classname DoWhileStatement }
   | for_kw:FOR init:(init:ForInit)?
     expr:(expr:Expr)? update:ForUpdate
     stmt:Block CLOSE_FOR
               { classname ForStatement }
   | break_kw:BREAK label:(label:Ident)?
               { classname BreakStatement }
```

```
   | continue_kw:CONTINUE label:(label:Ident)?
                { classname ContinueStatement }
   | return_kw:RETURN expr:(expr:Expr)?
                { classname ReturnStatement }
   | throw_kw:THROW expr:Expr
                { classname ThrowStatement }
   | synchronized_kw:SYNCHRONIZED expr:Expr block:Block
                { classname SynchronizedStatement }
   | try_kw:TRY block:Block body:TryBody
                { classname TryStatement }
   | assert_kw:ASSERT expr1:Expr expr2:(':' expr:Expr)?
                { classname AssertStatement }
   ;

/* Assignment has new optional keyword Set */
StatementExpr
   : SETVAR? expr:AssignmentExpr
                { classname AssignmentStatement }
   | expr:PreIncDecExpr
                { classname PreIncDecStatement }
   | expr:PostIncDecExpr
                { classname PostIncDecStatement }
   | expr:MethodCall
                { classname MethodCallStatement }
   | expr:InstanceCreate
                { classname InstanceCreateStatement }
   ;

/* Braces around SwitchBody are removed */
SwitchBlock
   : groups:(group:SwitchBlockGroup)*
       labels:(label:SwitchLabel)*
                { classname SwitchBody }
   ;

SwitchBlockGroup
   : labels:(label:SwitchLabel)+ stmts:(stmt:BlockStmt)+
                { classname SwitchGroup }
   ;

/* Colons removed. */
SwitchLabel
   : case_kw:CASE expr:Expr
                { classname CaseLabel }
```

```
   | default_kw:DEFAULT
               { classname DefaultLabel }
   ;

/* Comma separators removed from exprs list */
ForInit
   : exprs:(expr:StatementExpr)+
               { classname ForInitExprs }
   | decl:LocalVarDecl
               { classname ForInitExpr }
   ;

/* Comma separators removed from exprs list */
ForUpdate
   : exprs:(expr:StatementExpr)*
               { classname ForUpdateExprs }
   ;

TryBody
   : catches:(catch:Catch)+
               { classname CatchClauses }
   | catches:(catch:Catch)* finally:Finally
               { classname CatchFinallyClauses }
   ;

/* Parens surrounding param are removed. */
Catch
   : catch_kw:CATCH param:FormalParameter block:Block
               { classname CatchClause }
   ;

Finally
   : finally_kw:FINALLY block:Block
               { classname FinallyClause }
   ;

/* types */

PrimType: JBOOLEAN         { classname BooleanType }
   | JBYTE                 { classname ByteType }
   | JCHAR                 { classname JCharacterType }
   | JSHORT                { classname ShortType }
   | JINT                  { classname IntegerType }
   | JFLOAT                { classname FloatType }
```

```
   | JLONG                 { classname LongType }
   | JDOUBLE               { classname DoubleType }
   ;

Type
   : name:TypeName
               { classname SimpleType }
   | name:TypeName dims:(dim:Dim)+
               { classname ArrayType }
   ;

TypeName
   : type:PrimType     { classname PrimitiveType }
   | name:Name         { classname DefinedType }
   ;

ResultType
   : type:Type         { classname ExplicitResultType }
   | VOID              { classname VoidResultType }
   ;

/* names */

/* Dot is optional */
Name
   : id:Ident                  { classname SimpleName }
   | name:Name '.'? id:Ident   { classname QualifiedName }
   ;

/* Identifiers may be composed of several words strung together.
 * SingleName is translated to Java by concatenating each
 * identifier with no spaces in between. */
Ident
   : ids:(id:IDENTIFIER)+      { classname SingleName }
   ;

/* expressions */

NameOrPrimary
   : name:Name                 { classname NameExpression }
   | expr:Primary              { classname PrimaryExpression }
   ;

Primary
```

```
   : expr:OtherPrimary
                { classname OtherPrimaryExpression }
   | new_kw:NEW type:TypeName dExprs:(dExpr:DimExpr)+
     dims:(dim:Dim)*
                { classname NewArrayExpression }
   | new_kw:NEW type:TypeName dims:(dim:Dim)+ init:ArrayInit
                { classname NewArrayExpressionInit }
   ;

/* Dot is optional */
OtherPrimary
   : lit:IntLiteral
                { classname IntConstant }
   | lit:LongIntLiteral
                { classname LongIntConstant }
   | lit:StringLiteral
                { classname StringConstant }
   | lit:CharacterLiteral
                { classname CharacterConstant }
   | lit:FloatLiteral
                { classname FloatConstant }
   | lit:DoubleLiteral
                { classname DoubleConstant }
   | lit:TRUE_TOKEN
                { classname TrueConstant }
   | lit:FALSE_TOKEN
                { classname FalseConstant }
   | lit:NULL_TOKEN
                { classname NullConstant }
   | this_kw:THIS
                { classname ThisExpression }
   | lparen:'(' expr:Expr rparen:')'
                { classname ParenExpression }
   | expr:InstanceCreate
                { classname InstanceCreateExpression }
   | expr:FieldAccess
                { classname FieldAccessExpression }
   | expr:MethodCall
                { classname MethodCallExpression }
   | expr:ArrayAccess
                { classname ArrayAccessExpression }
   | name:Name '.'? this_kw:THIS
                { classname ClassAccessExpression }
   | type:ResultType '.'? class_kw:CLASS
```

```
                { classname ClassObjectExpression }
   ;

/* Comma separators removed from exprs list */
Args
   : exprs:(expr:Expr)+      { classname Arguments }
   ;

/* Parens removed from args list. Optional anonymous class
 * definition uses ClassBody2 instead of ClassBody. ClassBody
 * admits empty classes, ClassBody2 does not. NewExpression2
 * and EnclNewExpression2 added to support constructor calls
 * with no arguments. */
InstanceCreate
   : new_kw:NEW name:Name args:Args
       body:(body:ClassBody2)?
                { classname NewExpression }
   | new_kw:NEW name:Name NOARGS
       body:(body:ClassBody2)?
                { classname NewExpression2 }
   | expr:NameOrPrimary '.' new_kw:NEW id:Ident
       args:Args body:(body:ClassBody2)?
                { classname EnclNewExpression }
   | expr:NameOrPrimary '.' new_kw:NEW id:Ident
       NOARGS body:(body:ClassBody2)?
                { classname EnclNewExpression2 }
   ;

/* Three ways to say left bracket are supported. Right bracket
 * no longer allowed. */
DimExpr
   : lbracket:('[' | OFARRAY | SUB) expr:Expr
                { classname DimExpression }
   ;


Dim
   : lbracket:'[' rbracket:']'
                { classname Dimensions }
   ;

/* Dot is optional */
FieldAccess
   : object:Object '.'? field:Ident
```

```
                { classname ObjectFieldAccess }
   ;

/* Parens no longer surround args list. Dot is optional.
 * ThisMethodCall2 and OtherMethodCall2 are added to support
 * method calls with no arguments. */
MethodCall
   : name:Name args:Args
                { classname ThisMethodCall }
   | name:Name NOARGS
                { classname ThisMethodCall2 }
   | object:Object '.'? name:Ident args:Args
                { classname OtherMethodCall }
   | object:Object '.'? name:Ident NOARGS
                { classname OtherMethodCall2 }
   ;

/* Dot is optional */
Object
   : expr:Primary
                { classname PrimaryObject }
   | super_kw:SUPER
                { classname SuperObject }
   | name:Name '.'? super_kw:SUPER
                { classname EnclosingSuperObject }
   ;

/* Sub keyword added as alternative to left bracket. Right
 * bracket is no longer allowed. NameArrayAccessExpr2 and
 * PrimaryArrayAccessExpr2 added to support alternate phrasing
 * for array references. */
ArrayAccess
   : array:Name lbracket:('[' | SUB) index:Expr
                { classname NameArrayAccessExpr }
   | index:Expr ELEMENT OFARRAY array:Name
                { classname NameArrayAccessExpr2 }
   | array:OtherPrimary lbracket:('[' | SUB) index:Expr
                { classname PrimaryArrayAccessExpr }
   | index:Expr ELEMENT OFARRAY array:OtherPrimary
                { classname PrimaryArrayAccessExpr2 }
   ;

/* Cast operations have been rephrased. */
UnaryNoPMExpr
```

```
   : expr:NameOrPrimary
               { classname NameOrPrimaryExpression }
   | '!' expr:Expr
               { classname LogicalCompExpression }
   | '~' expr:Expr
               { classname BitwiseCompExpression }
   | expr:PostIncDecExpr
               { classname PostIncDecExpression }
   | JCAST expr:Expr TO name:PrimType dims:(dim:Dim)*
               { classname PrimTypeCastExpression
                 prec CAST }
   | JCAST expr:UnaryNoPMExpr TO name:Name dims:(dim:Dim)*
               { classname DefinedTypeCastExpression
                 prec CAST }
   ;

/* 'x' (times) is distinguished from '*' (star) in
 * MultiplicationExpression. Optional Set keyword added to
 * AssignmentExpression */
Expr
   : expr:UnaryNoPMExpr
               { classname ExpressionNoPlusMinus }
   | expr:PlusMinusExpr
               { classname PlusMinusExpression }
   | left:Expr 'x' right:Expr
               { classname MultiplicationExpression }
   | left:Expr '/' right:Expr
               { classname DivisionExpression }
   | left:Expr '%' right:Expr
               { classname RemainderExpression }
   | left:Expr '+' right:Expr
               { classname AdditionExpression }
   | left:Expr '-' right:Expr
               { classname SubtractionExpression }
   | left:Expr '<<' right:Expr
               { classname LeftShiftExpression }
   | left:Expr '>>' right:Expr
               { classname RightSignShiftExpression }
   | left:Expr '>>>' right:Expr
               { classname RightLogicShiftExpression }
   | left:Expr '<' right:Expr
               { classname LessThanExpression }
   | left:Expr '>' right:Expr
               { classname GreaterThanExpression }
```

```
   | left:Expr '>=' right:Expr
              { classname GreaterEqualExpression }
   | left:Expr '<=' right:Expr
              { classname LessEqualExpression }
   | left:Expr '==' right:Expr
              { classname EqualExpression }
   | left:Expr '!=' right:Expr
              { classname NotEqualExpression }
   | left:Expr '&' right:Expr
              { classname AndExpression }
   | left:Expr '^' right:Expr
              { classname ExclusiveOrExpression }
   | left:Expr '|' right:Expr
              { classname InclusiveOrExpression }
   | left:Expr '&&' right:Expr
              { classname ConditionalAndExpression }
   | left:Expr '||' right:Expr
              { classname ConditionalOrExpression }
   | expr:Expr instanceof_kw:INSTANCEOF type:Type
              { classname InstanceOfExpression }
   | expr:Expr '?' tExpr:Expr ':' fExpr:Expr
              { classname ConditionalExpression }
   | SETVAR? expr:AssignmentExpr
              { classname AssignmentExpression }
   ;

PlusMinusExpr
   : UPLUS expr:Expr
              { classname UnaryPlusExpression
                prec UPLUS }
   | UMINUS expr:Expr
              { classname UnaryMinusExpression
                prec UMINUS }
   | expr:PreIncDecExpr
              { classname IncDecPlusMinusExpression }
   ;

PreIncDecExpr
   : '++' expr:Expr      { classname PreIncExpression }
   | '--' expr:Expr      { classname PreDecExpression }
   ;

PostIncDecExpr
   : expr:UnaryNoPMExpr '++'
```

```
                { classname PostIncExpression
                  prec POSTINCR }
   | expr:UnaryNoPMExpr '--'
                { classname PostDecExpression
                  prec POSTDECR }
   ;

/* Alternate phrasing for AssignmentExpr added. */
AssignmentExpr
   : lhs:LeftHand (TO | '=') expr:Expr
                { classname AssignExpr }
   | lhs:LeftHand '+=' expr:Expr
                { classname PlusAssignExpr }
   | lhs:LeftHand '-=' expr:Expr
                { classname MinusAssignExpr }
   | lhs:LeftHand '*=' expr:Expr
                { classname TimesAssignExpr }
   | lhs:LeftHand '/=' expr:Expr
                { classname DivAssignExpr }
   | lhs:LeftHand '%=' expr:Expr
                { classname RemAssignExpr }
   | lhs:LeftHand '&=' expr:Expr
                { classname AndAssignExpr }
   | lhs:LeftHand '^=' expr:Expr
                { classname XorAssignExpr }
   | lhs:LeftHand '|=' expr:Expr
                { classname OrAssignExpr }
   | lhs:LeftHand '<<=' expr:Expr
                { classname LeftShiftAssignExpr }
   | lhs:LeftHand '>>=' expr:Expr
                { classname RightSignShiftAssignExpr }
   | lhs:LeftHand '>>>=' expr:Expr
                { classname RightLogicShiftAssignExpr }
   ;

LeftHand
   : name:Name              { classname LeftHandSideObject }
   | expr:FieldAccess       { classname LeftHandSideField }
   | expr:ArrayAccess       { classname LeftHandSideArray }
   ;

%%
```

Appendix C

XGLR Parser Algorithm

In this appendix, we present the XGLR parsing algorithm in its entirety, with support for ambiguous input streams and embedded languages. For an explanation of this algorithm and to see how it differs from GLR parsing, see Chapter 5.

XGLR-PARSE()
init *active-parsers list to parse state 0*
init *parsers-ready-to-act list to empty*
init *parsers-at-end list to empty*
init *lookahead-to-parse-state map to empty*
init *lookahead-to-shiftable-parse-states map to empty*
while *active-parsers list* $\neq \emptyset$
  PARSE-NEXT-SYMBOL(false)
copy *parsers-at-end list to active-parsers list*
clear *parsers-at-end list*
PARSE-NEXT-SYMBOL(true)
accept

PARSE-NEXT-SYMBOL(bool finish-up?)
SETUP-LEXER-STATES()
SETUP-LOOKAHEADS()
if *not finish-up?*
  FILTER-FINISHED-PARSERS()
  if *active-parsers list is empty?* **return**
init *shiftable-parse-states list to empty*
copy *active-parsers list to parsers-ready-to-act list*
while *parsers-ready-to-act list* $\neq \emptyset$
  remove *parse state p from list*
  DO-ACTIONS(p)

SHIFT-A-SYMBOL()

SETUP-LEXER-STATES()

for each *pair of parse states* p, q ∈ *active-parsers list*
  if *lexer state of* lex_p = *lexer state of* lex_q
    set lex_p *to* **copy** lex_q
for each *parse state* p ∈ *active-parsers list*
  let *langs* = *lexer-langs*[p]
  if |*langs*| > 1
    let *each of* q_1 .. q_n = **copy** *parse state* p
    for each *parse state* $q_i \in q_1$ .. q_n
      if $langs_i \neq$ *lexer language of* lex_p
        set *lex state of* lex_{q_i} *to* *init-state*[$langs_i$]
    add q_i *to* *active-parsers list*
  else if $langs_0 \neq$ *lexer language of* lex_p
    set *lexer state of* lex_p *to* *init-state*[$langs_0$]

SETUP-LOOKAHEADS()

for each *parse state*
    p ∈ *active-parsers list*
  set $lookahead_p$ *to first token lexed by* lex_p
  add <*offset of* $lookahead_p \times lookahead_p$> *to* *offset-to-lookaheads map*
  if $lookahead_p$ *is ambiguous*
    let *each of* q_1 .. q_n = **copy parse state** p
    for each *parse state* q ∈ q_1 .. q_n
      for each *alternative* a *from* $lookahead_p$
        set $lookahead_q$ *to* a
        add $lookahead_q$ *to equivalence class for* a
      add q *to* *active-parsers list*
for each *parse state* p ∈ *active-parsers list*
  add <$lookahead_p \times p$> *to* *lookahead-to-parse-state map*

FILTER-FINISHED-PARSERS()

for each *parse state* p ∈ *active-parsers list*
  if $lookahead_p$ = *end of input?*
    remove p *from* *active-parsers list*
    add p *to* *parsers-at-end list*

DO-ACTIONS(parse state p)

look up *actions*[p × $lookahead_p$]
for each *action*

```
  if action is SHIFT to state x
    add <p, x> to shiftable-parse-states
    add <lookahead_p×p> to lookahead-to-shiftable-parse-states map
  if action is REDUCE by rule y
    if rule y is accepting reduction
      if lookahead_p is end of input
        return
      if no parsers ready to act or shift or at end of input
        invoke error recovery
      return
    DO-REDUCTIONS(p, rule y)
    if no parsers ready to act or shift
      invoke error recovery and return
  if action is ERROR and no parsers
      ready to act or shift or at end of input
    invoke error recovery and return
```

DO-REDUCTIONS(parse state p, rule y)

```
for each equivalent parse state p⁻ below RHS(rule y) on a stack for parse state p
  let q = GOTO state for actions[p⁻×LHS(rule y)]
  if parse state q ∈ lookahead-to-parse-state[lookahead_p] and lookahead_q ≅ lookahead_p
      and (lookahead_p is end of input or lexer state of lex_q = lexer state of lex_p)
    if p⁻ is not immediately below q on stack for parse state q
      push q on stack p⁻
      for each parse state r such that r ∈ active-parsers list and r ∉parsers-ready-to-act list
        DO-LIMITED-REDUCTIONS(r)
  else
    create new parse state q with lookahead_p and copy of lex_p
    push q on stack p⁻
    add q to active-parsers list
    add q to parsers-ready-to-act list
    add <lookahead_q×q> to lookahead-to-parse-state map
```

DO-LIMITED-REDUCTIONS(parse state r)

```
look up actions[r×lookahead_r]
for each REDUCE by rule y action
  if rule y is not accepting reduction
    DO-REDUCTIONS(r, rule y)
```

SHIFT-A-SYMBOL()

```
clear active-parsers list
for each <p, x> ∈ shiftable-parse-states
```

```
if p is not an accepting parser
  if parse state x ∈ active-parsers list
    push x on stack p
  else
    create new parse state x with lookahead_p and copy of lex_p
    push x on stack p
    add x to active-parsers list
```

Appendix D

DeRemer and Pennello LALR(1) Lookahead Set Generation Algorithm

The author researched algorithms for computing lookahead sets for LALR(1) grammars. After the DeRemer and Pennello paper [19], there was a flurry of work by Park and Ives [72, 71, 73, 42, 43] to improve the algorithm's running time. After an academic debate was carried out in SIGPLAN Notices, the final word was given by Ives, in a letter to the editor in which he reconciled the differences between his algorithm and Park's and presented a final algorithm.

However, given that these works appeared in the early 1980's, they used fairly different terminology for parsing than today's students learn in their compiler courses. Worse, today's compiler courses teach nothing about any of the modern algorithms for generating LALR(1) parse tables, especially computing the lookahead sets. The Dragon book's last revision [2] does not include DeRemer and Pennello's work, and no compiler book that we have found explains the algorithm at all.

Why quibble with textbooks, if the algorithms are published in journals? Students wishing to learn how they work can just look them up. Unfortunately, that is not the case. The Park/Ives debate took place in SIGPLAN Notices which is an unrefereed publication. In fact, the last letter Ives wrote refers to a journal submission to explain his algorithm fully, but the article seems not to have appeared. Ives's letter uses terminology that perhaps people in the parsing community understood at the time, but which is completely undefined in the paper and cannot be found in any contemporary references (including the Dragon book).

We decided to implement the DeRemer and Pennello algorithm after failing to completely understand either the Park or the Ives algorithms. The improvements in running time achieved by the Park and Ives algorithms are no longer as important as they were; instead, we value algorithm

readability and reproducibility. Thus, we implemented the DeRemer and Pennello algorithm.

Unfortunately, DeRemer and Pennello's paper does not explain any of the data structures used within. Nor does it clearly indicate which elements of the graph are being operated upon in each phase of the algorithm. In fact, after a lengthy discussion in the paper of the algorithm's correctness, the authors never write down the final version of the algorithm, leaving readers to derive it on their own.

As a service to the community, we present a much more detailed version of the DeRemer and Pennello algorithm that can be implemented fairly simply. The algorithm and its data structures follow. Commentary on how each phase operates may be found in DeRemer and Pennello (their explanation once you know what you are supposed to be calculating is, in fact, excellent). Read this implementation with the paper in hand.

D.1 Data Structures

```
type reads-edge : {
   state : parse-state
   nt : nonterminal
   next-edges : set<reads-edge>
   depth-first-number : int
   lowlink : int
   in-edges : int
}

type includes-edge : {
   state : parse-state
   nt : nonterminal
   next-edges : set<includes-edge>
   depth-first-number : int
   lowlink : int
   in-edges : int
}

type goto-data : {
   nt : nonterminal
   state : parse-state
}

type shift-data : {
   term : terminal
   state : parse-state
```

}

type *symbol* : **union**<**terminal, nonterminal**>

type *parse-rule* : {
  LHS : **nonterminal**
  right-hand-side : **sequence**<**symbol**>
}

type *parse-item* : {
  rule : **parse-rule**
  dot : **int**
  lookbacks : **set**<**pair**<**parse-state, nonterminal**>>
  lookaheads : **set**<**terminal**>
}

type *parse-state* : {
  goto-table : **set**<**goto-data**>
  shift-table : **set**<**shift-data**>
  accepting-state? : **bool**
  read : **map**: **nonterminal** → **set**<**terminal**>
  follow : **map**: **nonterminal** → **set**<**terminal**>
  items : **set**<**parse-item**>
}

D.2 Global Variables

reads-edges-graph : **set**<**reads-edge**>
reads-roots : **set**<**reads-edge**>
reads-edges-stack : **stack**<**reads-edge**>
includes-edges-graph : **set**<**includes-edge**>
includes-roots : **set**<**includes-edge**>
includes-edges-stack : **stack**<**reads-edge**>
tarjan-num : **int**

D.3 Lookahead Set Computation Algorithm

compute-lookaheads()
reads-edges-graph ← ∅
reads-roots ← ∅
reads-edges-stack ← ∅
includes-edges-graph ← ∅

```
includes-roots ← ∅
includes-edges-stack ← ∅
tarjan-num ← 0
compute-reads()
compute-read()
compute-includes()
compute-follow()
compute-lookbacks()
compute-lookahead()
```

D.3.1 Compute Reads Set

```
compute-reads()
foreach state ∈ states
  compute-reads-for-state(state)
// toplogically sort reads edges
foreach edge ∈ reads-edges-graph
  if edge.in-edges = 0
    reads-roots.insert(edge)
```

```
compute-reads-for-state(p : parse-state)
// reads(p : state, A : nonterminal) = (r : state, C : nonterminal)
// if p → r via A ∧ r → s via C ∧ C ⇒* ϵ
foreach goto ∈ p.goto-table
  goto-NT ← goto.nt
  r ← goto.state
  foreach goto-next ∈ r.goto-table
    goto-next-NT ← goto-next.nt
    if goto-next-NT.is-nullable?()
      from-edge ← get-reads-edge(p, goto-NT)
      to-edge ← get-reads-edge(r, goto-next-NT)
      from-edge.next-edges.insert(to-edge)
      to-edge.in-edges ← to-edge.in-edges + 1
```

```
get-reads-edge(state : parse-state, nt : nonterminal)
look up reads-edge(state, nt) ∈ reads-edge-graph
if found
  return reads-edge
else
  return new reads-edge(state, nt)
```

```
compute-read()
// F state = F' state
foreach state ∈ states
  foreach goto ∈ state.goto-table
    goto-NT ← goto.nt
    r ← goto.state
    foreach shift ∈ r.shift-table
      state.read[goto-NT].insert(shift.term)
    if r.accepting-state?
      state.read[goto-NT].insert(eofTerminal)
foreach edge ∈ reads-edge-graph
  edge.depth-first-number ← 0
tarjan-num ← 0
reads-edge-stack ← ∅
// make sure to do the roots of the graph first
foreach edge ∈ reads-edge-roots
  if edge.depth-first-number = 0
    tarjan-read(state, edge)
// just in case we missed any that are disconnected from the graph
foreach edge ∈ reads-edge-graph
  if edge.depth-first-number = 0
    tarjan-read(state, edge)


tarjan-read(state : parse-state, edge : reads-edge)
edge.depth-first-number ← ++tarjan-num
reads-edge-stack.push(edge)
lowlink ← |reads-edge-stack|
foreach next-edge ∈ edge.next-edges
  if next-edge.depth-first-number = 0
    tarjan-read(state, next-edge)
  if lowlink ≥ next-edge.lowlink
    lowlink ∈ next-edge.lowlink
  edge.state.read[edge.nt].insert(next-edge.state.read[next-edge.nt])
if lowlink = |reads-edge-stack|
  // found a cycle
  next-edge ← reads-edge-stack.pop()
  while next-edge ≠ edge
    next-edge.lowlink ← ∞
    next-edge.state.read[next-edge.nt].insert(edge.state.read[edge.nt])
    next-edge ← reads-edge-stack.pop()
```

D.3.2 Compute Includes Set

```
compute-includes()
foreach state ∈ states
  compute-includes-for-state(state)
// toplogically sort includes edges
foreach edge ∈ includes-edges-graph
  if edge.in-edges = 0
    includes-roots.insert(edge)


compute-includes-for-state(state : parse-state)
// includes(p : parse-state, A : nonterminal) = (p' : parse-state, B : nonterminal)
// if B → β A γ, γ ⇒* ϵ ∧ p' → p via β
foreach goto ∈ state.goto-table
  goto-NT ← goto.nt
  p ← state
  foreach rule ∈ get-rules-for-nonterminal(goto-NT)
    was-nullable-after-dot? ← false
    dot ← 0
    foreach symbol ∈ rule.right-hand-side
      dot ← dot + 1
      if is-nonterminal?(symbol)
        if was-nullable-after-dot? ∨ is-nullable-after-dot?(rule, dot)
          from-edge ← get-includes-edge(p, symbol)
          to-edge ← get-includes-edge(state, goto-NT)
          from-edge.next-edges.insert(to-edge)
          to-edge.in-edges ← to-edge.in-edges + 1
          was-nullable-after-dot? ← true
        p ← p.getGoto(symbol)


get-includes-edge(state : parse-state, nt : nonterminal)
look up includes-edge(state, nt) ∈ includes-edge-graph
if found
  return includes-edge
else
  return new includes-edge(state, nt)
```

D.3.3 Compute Follow Set

```
compute-follow()
// F state = F' state
foreach state ∈ states
```

```
  foreach goto ∈ state.goto-table
    goto-NT ← goto.nt
    r ← goto.state
    state.follow[goto-NT.insert(state.read[goto-NT])
foreach edge ∈ includes-edge-graph
  edge.depth-first-number ← 0
tarjan-num ← 0
includes-edge-stack ← ∅
// make sure to do the roots of the graph first
foreach edge ∈ includes-edge-roots
  if edge.depth-first-number = 0
    tarjan-follow(state, edge)
// just in case we missed any that are disconnected from the graph
foreach edge ∈ includes-edge-graph
  if edge.depth-first-number = 0
    tarjan-follow(state, edge)


tarjan-follow(state : parse-state, edge : reads-edge)
edge.depth-first-number ← ++tarjan-num
includes-edge-stack.push(edge)
lowlink ← |includes-edge-stack|
foreach next-edge ∈ edge.next-edges
  if next-edge.depth-first-number = 0
    tarjan-follow(state, next-edge)
  if lowlink ≥ next-edge.lowlink
    lowlink ∈ next-edge.lowlink
  edge.state.follow[edge.nt].insert(next-edge.state.follow[next-edge.nt])
if lowlink = |includes-edge-stack|
  // found a cycle
  next-edge ← includes-edge-stack.pop()
  while next-edge ≠ edge
    next-edge.lowlink ← ∞
    next-edge.state.follow[next-edge.nt].insert(edge.state.follow[edge.nt])
    next-edge ← includes-edge-stack.pop()
```

D.3.4 Compute Lookbacks and Lookaheads

```
compute-lookbacks()
foreach state ∈ states
  foreach item ∈ state.item-set
    next:
    if item.is-dot-at-beginning?
      next-state ← state
```

```
        rule ← item.rule
        if not is-epsilon?(rule)
          foreach symbol ∈ rule.right-hand-side
            next-state ← next.get-state-after-goto(symbol)
            if next-state.is-accepting-state?
              continue next:
        next-item ← next.get-item-with-rule-and-dot-at-end(rule)
        next-item.lookbacks.insert(pair<state, rule.LHS>)


compute-lookahead()
foreach state ∈ states
  if state.is-accepting-state?
    continue
  foreach item ∈ state.item-set
    if item.is-dot-at-end?
      // only do lookaheads for reducing items
      foreach lookback ∈ item.lookbacks
        next-state ← lookback.first
        next-NT ← lookback.second
        item.lookaheads.insert(next-state.follow[next-NT])
```