A Java Programming Tool for Students with Visual Disabilities

Ann C. Smith
Computer Science Dept.
Saint Mary's University
Winona, MN 55987
507/457-1430
asmith@cs.smumn.edu

Joan M. Francioni
Computer Science Dept.
Winona State University
Winona, MN 55987
507/457-2336
joanf@wind.winona.msus.edu

Sam D. Matzek
Computer Science Dept.
Saint Mary's University
Winona, MN 55987
507/457-1430
sdmatzek@cs.smumn.edu

ABSTRACT

This paper reports on a tool for assisting students with visual disabilities in learning how to program. The tool is meant to be used by computer science majors learning the programming language Java. As part of the developmental process of building this tool, we have implemented a rapid prototype to be used by people with disabilities in order to define appropriate requirements for the full version of the tool. This requires that the prototype is completely usable via a keyboard and speech interface, and it is easily adaptable for trying out different strategies. In this paper, we present the motivation and philosophy of the full tool, called *JavaSpeak*. We also present the details of a prototype implementation of JavaSpeak.

Keywords

Java, programming tool, students with visual disabilities, learning to program

INTRODUCTION

Nationally, the trend for the number of Computer Science majors at the college level has increased significantly over the past five years. [7] On the other hand, there are very few students with visual disabilities among these large numbers of computer science majors – even though computer science is currently a popular major choice for high school students with disabilities planning to go to college [3]. This situation is not because the subject matter is inherently unlearnable by students with visual disabilities. Rather it is because of barriers that exist for these students to study computer science in a traditional computer science curriculum. For example, one barrier is simply the way in which most computer science courses are taught – a way that makes heavy use of visual images and abstractions. This is the case in the classroom as well as in textbooks. Another barrier is that the science and mathematics courses of a computer science curriculum are usually not set up to use assistive technology that is specifically designed for these kinds of classes. The technology exists; it is just not readily available in most universities. A third barrier for students with visual disabilities comes into play in actually *learning to program*.

Paper published in proceedings of *ACM Assets 2000*, Washington, D.C., November 2000, pp. 142-148.

Learning to program is a major part of being a computer science student and, as such, is the main emphasis of the first couple of years in a computer science curriculum. It is a challenging task for many students, with or without any kind of disability, and many tools and techniques have been developed to assist students in learning this skill. In essence, students have to learn how to both formulate an algorithmic solution to a problem and then translate that solution into a semantically equivalent program, which is also syntactically correct. As they develop their programs, they need to be able to group individual tokens (words and symbols) of the language together as syntactic units. These syntactic units are then further grouped together to form larger syntactic units. As the program grows, it must be understood both in the small and in the large.

For students with visual disabilities, these tasks take on an extra level of difficulty. Using existing screen readers, they can have their programs spoken out loud for them. However, they must still process everything in the program sequentially. This is somewhat equivalent to trying to write or read this paper without any newlines, blank lines, section headings, page breaks, or special character fonts. If you can get it all in the first pass, great. But having to go back and find a part of a sentence that you have already written or read is not an easy task.

We are involved in a long-term project to try and make the computer science curriculum accessible to students with visual disabilities by addressing, in particular, the three

barriers named above. As part of our work to teach students with visual disabilities how to program, we are developing a specialized programming environment for the Java programming language, called *JavaSpeak*.

Basically, JavaSpeak is an editor with aural feedback designed to provide a user with useful information about a program's structure and semantics. It is designed to parse the program and "speak" the program's structure to a blind user, in much the same way that separate lines and indentation and color all help to "show" the structure of a program to a sighted user. In this paper, we present the design philosophy for this tool and give the details of a prototype version of JavaSpeak.

The paper is organized as follows. First we discuss other research related to this work. An overview of JavaSpeak and the details of the prototype follow this. Future plans are then discussed, followed by the concluding remarks section.

BACKGROUND

In his book on auditory user interfaces (AUI), Raman describes a speech-enabling approach to developing applications that separates information and computation from the user interface [15]. When this is done, it is possible for different user interfaces to portray the functionality of the application in a manner most suited to a given user environment. In other words, if the information and computation is available separate from the user interface, the user interface is free to present that information in either a graphical way or in an auditory way – or even in any other way.

This approach has been taken by a number of tools for increasing the accessibility of underlying applications. Raman, himself, used this approach for both AsTeR [13], a system for producing aural renderings of documents written in LaTeX, and Emacspeak [14], a tool that extends the Emacs editor to provide a "speech-enabled desktop" (see subsection on Emacspeak below). With these two programs, he demonstrated early on the benefits of integrating speech into the human-computer interaction as opposed to simply generating a spoken version of visual output. Commercial tools like Jaws for Windows [9] and Outspoken [12] have since extended this concept for navigating Microsoft Windows environments.

HTML documents on the web make use of this approach as well by maintaining a clear separation between information and the user interface. Graphical browsers use the HTML tags to express information in a way that makes sense visually. Similarly, auditory web browsers can use the HTML tags to govern how to present the same information in a way that makes sense aurally. Emacspeak includes such a speech interface for web browsing through Emacs. Two other examples of stand-alone auditory systems for Web access are IBM's Home Page Reader [5] and the BrookesTalk [16] tool. In addition to presenting the basic text in a spoken version, each of these tools offer ways for the user to effectively navigate through the web page. For example, Home Page Reader provides a "Where am I?" command that tells the location of an element on the current page. BrookesTalk has a page summarization feature that analyzes the page information and forms an abstract of the page approximately $1/5^{th}$ the size of the original page. All three tools use different voices to give different cognitive cues. The techniques developed for these and other auditory web tools, as well as those developed for tools that speak mathematical equations (e.g., MAVIS [10]), offer a rich new vocabulary for AUIs that can be used in JavaSpeak.

Little formal research has been done on the topic of assisting people with visual disabilities in learning to program. One exception to this is a project led by Ivan Kopecek at the Faculty of Informatics in Czech Republic [11]. The project is designed around the logic programming language Prolog. In logic programming, the computer is provided with a specification of the problem in the form of facts and rules, as opposed to a sequence of steps reflecting a specific solution to the problem. The conjecture is that people with visual disabilities may be able to learn this style of programming more easily than imperative or object-oriented programming. Since our project is related to students learning an object-oriented language, we can not use this work directly. However, some of their experiences gained in working with blind programmers will be beneficial.

We have done preliminary interviews with blind programmers to gain some insight into how they program. By far, the most common technique mentioned is the use of comments to mark the beginnings and ends of syntactic structures. The second most common thing mentioned is the use of meaningful identifier names.

Emacspeak

Emacspeak provides a complete speech interface to a number of user applications. In particular, it provides functions geared directly to programming, such as speaking program code in a meaningful way, browsing source code, and interfacing with the compiler error output and the debugger. For someone who knows how to program already and is familiar with a Unix environment, this tool provides a very effective eyes-free environment.

The main difference between Emacspeak and JavaSpeak is that JavaSpeak is designed, a-priori, as a tool for students who are just learning to program. As such, the goal is for the tool to help students *learn* the connection between the syntax of the language and the semantics of the language. JavaSpeak will make use of some of the techniques introduced in Emacspeak but, in many ways, it will not be as powerful as Emacspeak for experienced programmers.

But for students who are just learning to program, JavaSpeak will provide feedback appropriate to their level of understanding in such a way as to hopefully build upon this.

DEVELOPMENT PROCESS FOR JAVASPEAK

The speech-enabling approach of Raman's described above is the approach that we are following with JavaSpeak. As such, we need to first identify the information that is important to be presented. Only then can we define an effective way of aurally presenting the information.

As computer science educators for a number of years, we have a large amount of experience working with students who are learning to program. We do not, however, have much experience working with students who have visual disabilities. So, although we have basic pedagogical knowledge of what kind of information should be presented to help a student learn to program, we do not know if this information is exactly what a blind student needs to learn how to program. Therefore, we need to work directly with students and programmers who have visual disabilities to figure this out.

Our approach to this problem has been to develop a rapid prototype of JavaSpeak, with a very flexible implementation, that can be used as a mechanism for gaining user feedback. After users with visual disabilities have had a chance to experiment with the prototype, their feedback will then drive the development of the full set of functional specifications for the JavaSpeak tool.

Initial Pedagogical Requirements for JavaSpeak

To learn how to program, students first learn the simple syntactic units in the language. In general, this is not very difficult for them to do. It is when students start combining simple syntactic units together to form larger syntactic units that they run into trouble. For example, consider the simple *if-else* statement below:

```
if (x == 0)
System.out.println("Done");
else System.out.println("Continue");
```

Students who understand the basic idea of a conditional can learn an *if-else* statement fairly quickly, even when they don't use any indentation. But consider combining two *if* statements with only one *else* part as shown below:

```
1.  if (x == 0)
2.  if (y == 0)
3.  System.out.println("Done");
4.  else x = y;
```

Understanding whether the *else* statement at line 4 goes with the *if* at line 1 or the *if* at line 2 is a much harder concept for students to figure out. (A note to non-programmers: no matter how the statements are indented, the *else* of line 4 goes with the *if* of line 2.)

In addition to students learning how to write correct programs in the sense that the right answers are computed, they must also learn how to design and organize their programs in an appropriate way. Java programs are organized around object-oriented and traditional structured programming paradigms. Even though students start with small programs in the early classes, we still enforce organizational concepts from the beginning. Nonetheless, it is sometimes difficult for them to learn these concepts.

The above discussion illustrates two different foci of learning to program: the *syntactic structure* of the program and the *organizational structure* of the program. The syntactic structure category is related to how program components work together and is based directly on the definition of the programming language. The organizational structure category is related to how the user as a human arranges the code to keep track of what's going on. Incorrect syntactic structure should signal to the user that errors were made during the algorithm to language translation phase of program development. An organizational structure that is difficult to decipher should signal to the user that there were errors made during the algorithm development or design phase of program development.

To help students learn how to program, we want to provide pedagogical information about both the syntactic and the organizational structures of their programs. In Table 1, we have identified a set of seven basic kinds of information related to these two structures that JavaSpeak should be able to represent aurally to the user. For clarification, the table includes example visual techniques that are often used in programming environments to depict the named concept. Numbers 1-4 are related to the syntactic structure of the program; numbers 5-7 to the organizational structure.

Kind of Information	Indicates	Visual Presentation Techniques
1. Phrasing	structure of a syntactic unit	newlines; language punctuation; color
2. Blocks	groups of statements that are logically related	indentation; blank lines
3. Nesting Levels	nested hierarchy of syntactic units	cascading indentation; newlines
4. Differentiation	reserved words, special kinds of identifiers	color; font; naming conventions (e.g., capitalized class names, upper case symbolic constants)
5. Identifier Names	meaningful names	mixed case for multiple words (e.g. FirstName)
6. Background	parts not currently being considered	compacting text; color and font; alignment; special character combinations (e.g., //)
7. Constituent Parts	data members vs. methods, composition, inheritance hierarchy	spatial conventions; heading separation characters (e.g., /**********/)

Table 1. Pedagogical Information Required of Prototype

JAVASPEAK PROTOTYPE

The intent of the full JavaSpeak tool is to generate auditory contextual cues about a program, which are based on the definition of the Java programming language, that will assist visually impaired students in learning how to program. The JavaSpeak *prototype's* main requirement, therefore, is to maximize our ability to gain user feedback about ways in which auditory cues can support this process. In this section, we describe the details of the working JavaSpeak prototype, with which we are starting our usability tests.

Prototype Requirements

There are three primary requirements for the JavaSpeak prototype. Specifically, the prototype must

(1) Be completely usable by someone without sight. This implies both the use of the tool and navigation within the program must be accessible via a keyboard and speech interface.

(2) Capture compiler-related information about a program that facilitates an aural rendering of the information listed in Table 1.

(3) Be designed to be easily extensible and adaptable. In particular, the tool must support experimentation with different configurations and different aural renderings of a program's structure.

Prototype Design

The JavaSpeak prototype has the look of a traditional GUI program editor with a text area and a drop down menu system. There are five menu options: File, Edit, Focus, Reader, and Help. Table 2 shows the choices available under each menu option.

File	Edit	Focus	Reader	Help
New	Cut	Select Text	1. Basic Compilation Unit	Contents
Open	Copy	Set Range	2. Data Members	
Save	Paste	Stop Reading	3. Method Names	
Save As	Select All		4. Full Method Data	
Options			5. Block Statements	
Exit			6. Full Block Data	
			7. Token	
			8. Character	

Table 2. Menu Options of Editor

The File and Edit options provide basic word processing capability. The Focus options provide ways to select the part of the program to be read and to stop the reading of the program. The Reader options start an aural rendering of the selected part of the program at one of several pre-defined levels of syntactic granularity. The Help option provides assistance with both JavaSpeak tool usage and introductory level Java programming in general.

The design of the prototype is divided into three main parts: the navigational subsystem, the syntactic reader subsystem, and the aural cue functions.

Navigational Subsystem

Tool navigation functions and menu option selection are both function-key driven. The function-key mappings we use currently model those used in Jaws for Windows [9]. In addition to the standard word processing navigation functions, users are provided with the ability to select a subset of their program for the aural rendering. The focus of the rendering can be defined to go from the beginning to the end of the program, to start at a specific line number, or to only render a range of line numbers.

Syntactic Reader Subsystem

The syntactic reader subsystem (SRS) is central to JavaSpeak as a teaching tool. It provides the user with aural cues to help make the conceptual connection between the written text and the definition and structure of the Java language. The basic strategy of the SRS is to parse a given program and generate an aural rendering of the program that is then passed to IBM's ViaVoice [6] program for the actual speech output.

The rendering of aural cues by the SRS is a configurable parameter in the JavaSpeak prototype. Cues can be given in many forms (see below) and can serve to highlight any level of program structure granularity, from the smallest granularity (token by token), to a mid-level granularity (e.g., block statement), to the largest granularity (compilation unit). Table 3 gives an example of a section of code and the renderings of this code at three different granularity levels. (Only the first part of the rendering for Level 7 is shown.) A desired outcome from the experimental use of the prototype is to gain a better understanding of which combinations of aural forms and program levels provide the user with the most useful contextual information.

Java Code Segment	Level 1 Compilation Unit	Level 5 Block Statement	Level 7 Tokens

`public class MyClass {`
`  String x;`
`  int test(String s) {`
`  System.out.println(“hi”);`
`  if (x == s) {`
`  for(int i=0; i<2; i++) {`
`  System.out.print(“*”);`
`  System.out.print(“-“);`
`  }`
`  System.out.println();`
`  }`
`  }`
`}`
	begin class declaration
	public class my class
	end class my class
		begin class declaration
		public class my class
		begin method test
		begin if
		begin for
		end for
		end if
		end method test
		end class my class
			begin class declaration
			public class my class
			string x
			begin method int test
			begin parameters
			string s
			end parameters
			open brace
			system dot out dot print line, open parenthesis, open quote, hi, close quote, close parenthesis
			begin if
			if x is equal to s
			open brace
			begin for
			for int i assigned 0, i less than 2, i plus plus
			open brace
			. . .

Table 3. Renderings of Code Segment at Different Granularities

Aural Cue Functions

Aural cue functions include a variety of techniques such as text-insertion, symbol-to-text-substitution, text-to-alternate-text-substitution, and alteration of voice, tone, and other sound characteristics. Following are brief descriptions of each of these techniques along with examples of the aural cues that may be produced by each technique.

Text-insertion involves feeding additional text to the speech reader at key syntactic points. For example, the text “begin while expression” and “end while expression" can be added to the aural rendering of a while loop. The added text has the effect of emphasizing the loop test while distinguishing it from the loop body. This emphasis not only serves to reinforce the generalized concept of a loop’s test as separate from its body, but also maps nicely to the language that we, as educators, use to describe loop execution flow.

Symbol-to-text-substitution is about storing a replacement string in place of each occurrence of the symbol itself in the text string to be fed to the speech reader. This technique provides the users with a cue as to how the operator is used. For example, the symbol-to-text-substitution of the symbols “= =” to the text “is equal to” or of the symbol “=” to the text “is assigned” helps to clarify the difference in meaning between these frequently interchanged operators.

Text-to-alternate-text-substitution is basically the same as symbol-to-text-substitution. An example of its use is in the text substitution of the function call phrase “System.out.println()” with the text “system dot out dot print line.” The resulting aural rendition matches the language that would typically be used by educators to name this function call. Therefore this particular text substitution cues the user into the connection between the written text and common Java terminology.

Although each of these examples uses some sort of text replacement, aural cues can be based on a character reading. This is necessary for a student to be able to relate the actual symbols (e.g., "= =") to their meaning (e.g., "is equal to").

In addition to the above arual cue functions, we will also experiment with different approaches to "syntax coloring" [15]. By altering speech tone and emphasis, the user can be alerted to special tokens (e.g., reserved words), spacing, commenting, and syntactic units. For example, comments can be read using a different voice than program code, method names can be read with a different voice than data members, and reserved words can be emphasized by lowering or elevating the pitch. This is the same sort of strategy used in auditory web browsers that use male and female voices to distinguish between links and non-links on a Web page.

Prototype Implementation

Two existing tools were used extensively in the implementation of the syntactic reader subsystem: the Java Compiler Compiler (JavaCC) [8] and IBM’s ViaVoice. JavaCC was used to generate code for a Java parser that could then be modified as necessary to generate aural renderings of a parsed program. ViaVoice, which has a published API, was used as our speech reader.

A diagram showing the overall design of the syntactic reader subsystem is shown in Figure 1. The `Generator` class is an abstract class that provides an interface of accessible functions to the outside world. All Generator classes contain a `generate(String)` method that takes in a string containing the Java code for which the user wishes to generate spoken output. The `generate()` method then returns another string representing an aural rendering of the code, which is passed back to the main program. In particular, LexGenerator generates a rendering based solely on the tokens of the input, and SyntaxGenerator generates one of several renderings at levels based on the syntactical structure of the program.

Generator has a TokenReplacer class that manages replacements for tokens. TokenReplacer classes contain a `getReplacement(String)` method that returns the replacement for the token in String.

JavaCC was used to generate the SyntaxParser, the AST (abstract syntax tree) classes, and various utility classes including SyntaxParserTokenManager. LexGenerator uses the SyntaxParserTokenManager to get the tokens from the string passed to the `generate()` method. The tokens are then passed to the PhoneticTokenReplacer, after which they are appended to the string to be output.

The SyntaxGenerator is a little more complex. The SyntaxGenerator's `generate()` method passes the input string to the SyntaxParser, which generates a syntax tree based on the structure of the code. The `generate()` method then uses the SyntaxVisitor on the generated abstract syntax tree. The SyntaxVisitor traverses the tree and generates the correct rendering for a given level. The SyntaxVisitor also filters the tokens in the input string through a PhoneticTokenReplacer. It then returns the rendering as a string to the SyntaxGenerator, which in turn returns the string to the rest of the program.

Under the current (05/00) implementation, text returned by the generators to the main program is passed to a TextReader class (not shown) that uses the Java Speech API to read the text in a separate thread.

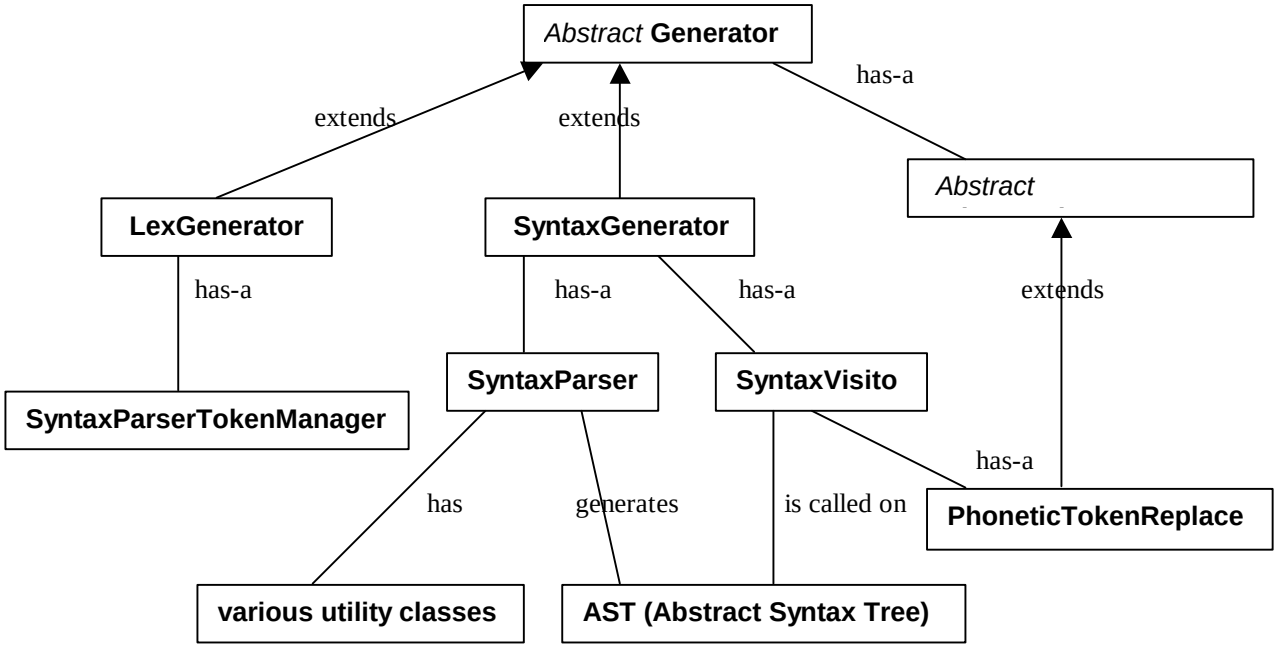

Figure 1. Syntactic Reader Subsystem

The technique of using different voice tones and emphases while rendering program text is implemented by inserting JSML (Java Speech Markup Language) tags into the text stream to be read by ViaVoice.

NEXT PHASES OF DEVELOPMENT

The immediate next phase of development for this project will be to work with programmers who have a visual disability in using the JavaSpeak prototype. [Information about this effort can be found at the JavaSpeak web site, http://cs.smumn.edu/csmap/javaspeak.html.] Based on these tests, and working in conjunction with the programmers, we will define a first set of requirements for the full JavaSpeak. In [2] it is pointed out that "Blind users of graphical user interfaces are especially affected by arbitrary violations of design guidelines with respect to application layout, behavior, and key mappings." As much as possible, we will define JavaSpeak to be consistent with other tools developed for blind and visually impaired users.

We are developing JavaSpeak to be used by students in introductory computer science classes, primarily CS1, CS2, and Data Structures. When we have the next version of JavaSpeak developed so that it can be used in real time by visually impaired students taking CS1, we will get an even clearer idea of what should and shouldn't be included in the tool. (As part of our long-term project, we have funds for recruiting and supporting students with visual disabilities in our computer science programs.)

The use of non-verbal cues in JavaSpeak is an area of future exploration. A number of existing assistive tools make effective use of non-verbal cues to varying degrees. There has also been work done in the area of using non-verbal sounds to represent the behavior of a program in execution (e.g., [4] and [1]). This too will be explored.

It will be very useful to students for JavaSpeak to give interactive feedback and to support debugging activities. We plan to work on both of these extensions.

CONCLUDING REMARKS

Our focus is on teaching students to learn how to program. Therefore, we started out with our understanding as teachers of how students learn to program. The information we want to present is related to what will help students learn the Java programming language as well as to help them develop the art/skill of programming. As such, we believe the tool can help students without visual disabilities as well.

REFERENCES

1. ACM Special Interest Group on Sound and Computation, http://www.acm.org/sigsound
2. Bergman, Eric and Johnson, Earl, "Towards Accessible Human-Computer Interaction," in *Advances in Human-Computer Interaction*, Vol. 5, edited by Jakob Nielsen, 1995. Available at http://www.sun.com/access/updt.HCI.advance.html

3. Blackorby, J., Cameto, R., Lewis, A., & Hebbeler, K., "Study of Persons with Disabilities in Science, Mathematics, Engineering, and Technology," SRI International, Menlo Park, CA, 1997.
4. Francioni, Joan and Jackson, Jay, "Breaking the Silence: Auralization of Parallel Program Behavior," in *Journal of Parallel and Distributed Computing*, June 1993.
5. IBM, Home Page Reader, Available at http://www.austin.ibm.com/sns/hpr.html
6. IBM, ViaVoice, Available at http://www-4.ibm.com/software/speech/
7. Irwin, Mary Jane and Friedman, Frank, "1998-1999 CRA Taulbee Survey," in *CRN*, publication of Computing Research Association, March 2000. Available at http://www.cra.org/CRN/online.html
8. JavaCC, The Java Parser Generator, Available at http://www.metamata.com/JavaCC/
9. Jaws for Windows, http://www.hj.com/
10. Karshmer, Arthur, *MAVIS (Mathematics Accessible to Visually Impaired Students)*, New Mexico State University, http://www.nmsu.edu/~mavis
11. Kopecek, Ivan, *Programming for Visually Impaired People*, http://www.fi.muni.cz/~kopecek/pvip.htm
12. outSPOKEN, http://www.humanware.com/E/E2/E2E.html
13. Raman, T. V., "AsTeR – Toward Modality-Independent Electronic Documents," *DAGS 95*, 1995, http://cs.cornell.edu/home/raman/publications/dags-95/paper.html
14. Raman, T. V., "Emacspeak – Direct Speech Access," in *Proceedings of the Second Annual ACM Conference on Assistive Technologies, Assets '96*, April 11 - 12, 1996, Vancouver Canada, pp. 32-36. http://cs.cornell.edu/home/raman/emacspeak/publications/assets-96.html
15. Raman, T. V., *Auditory User Interfaces: Toward the Speaking Computer*, Kluwer Academic Publishers, Boston, 1997.
16. Zajicek M., "Increased Accessibility to Standard Web Browsing Software for Visually Impaired Users," ICCHP, 2000. Available via BrookesTalk home page, http://www.brookes.ac.uk/schools/cms/research/speech/btalk.htm