A System for Wizard of Oz Studies in Natural Language Programming

Dana B. Dahlstrom

2001-08-31
1 Abstract

In a Wizard of Oz study, subjects are intended to believe they are using software when in fact there is a person—called the “wizard”—behind the scene playing the role of the program. This paper describes a system designed for such a study where the program in question is a natural language interface for computer programming.

The requirements for the system are outlined, and the breakdown of the tasks and their solutions are presented. The interfaces for the user and the wizard and the motivations for their design are detailed. The method for keeping a useful record of the sessions is explained.

Finally, the procedure for running a session with the system is given, and the performance of the system in a live study is reviewed and its strengths and weaknesses are considered.
Contents

1 Abstract .............................................. 1
2 Introduction ......................................... 3
3 Requirements Analysis ......................... 4
4 The Audio ........................................... 5
5 The User’s Interface .............................. 6
6 Handling the Speech Recognition Output .. 8
7 The Wizard’s Interface ......................... 9
8 Sending Files to the User ...................... 10
9 Collating the Data ............................... 11
10 Running a Session ............................... 12
11 Conclusions ....................................... 13
12 Acknowledgements ............................ 14
A Appendix ........................................... 16
   A.1 wizard.el: Wizard Code Window (Emacs Lisp) ......... 16
   A.2 msg.el: Wizard Message Window (Emacs Lisp) ........ 19
   A.3 user.bat: User Start Script (DOS Batch) ............... 22
   A.4 startsession.sh: Wizard Start Script (Bourne Shell) .. 22
   A.5 endsession.sh: Wizard End Script (Bourne Shell) .. 23
   A.6 sendfiles.pl: File Mailing Script (Perl) ............... 24
   A.7 collate.pl: Data Collation Script (Perl) ............... 26
   A.8 pcmtee.c: PCM Tee Program (C) ..................... 28
2 Introduction

In his 1950 paper “Computing Machinery and Intelligence”, Alan Turing proposed to evaluate the question of whether machines can “think” by playing the “imitation game”, in which a human interrogator is challenged to distinguish between a machine and another person by asking questions of each [5]. In what has since been called the Turing Test, if the interrogator cannot correctly identify which entity is the real person, then the machine can be said to possess intelligence.

Our task was not to fool someone into believing a machine was a person, but the reverse: to disguise responses typed by a person as output generated by a computer program. We wanted to do this to facilitate a “Wizard of Oz” study, in which we would examine how users interacted with a software system which had not yet been developed. In fact, we plan to use the data gathered by the study—which is in progress as of this writing—in the design of an actual system.

The motivations for using a Wizard of Oz approach are laid out in more detail in [1]; the technique has been used to mock up programming by demonstration systems [3], and general prototyping tools for building speech interfaces have been created [2].

The system we wanted to simulate is a tool to write computer programs through a speech interface. The system should recognize verbal commands which are as abstract as possible, freeing the programmer not only from typing but from dictating programming language syntax, which is all the more cumbersome when it must be spoken aloud.

For example, the user might say, “Create a function called ‘fact’ which accepts an integer called ‘n’ and returns an integer.” In response, the system might output the following C/C++ code:

int fact(int n) {
}

With current speech recognition and natural language processing techniques, such a system would be far from perfect at recognizing its users’ intentions, even if their speech were clear and their commands unambiguous. More to the point, constructing the system would be a major undertaking. The Wizard of Oz technique has allowed us to do user testing with a better recognition system than is currently possible, and without actually implementing it in software.
Perhaps the biggest barrier to an actual system of this kind is inaccuracy in speech recognition. The natural language processing part of the equation is already approaching a usable system [4].

3 Requirements Analysis

To maintain the illusion a computer was doing the work, we had to keep our wizard “behind the curtain”—undetectable by the user. The wizard and the user would have to be in different rooms, sonically isolated from each other. Preferably, the rooms should be far enough apart to prevent accidents like the user seeing a person going into another room before the beginning of each session.

The wizard and the user would communicate, but only through specific channels which would seem mechanical to the user. We needed to consider every aspect of the interaction to make sure we gave no obvious clues about the true nature of the system.

The wizard would have to hear the user’s voice at all times, without the user being aware someone was listening. Since actual speech recognition software would need sound from a microphone too, we could use one with no worry. In fact, since speech recognition software generally requires a clean sound signal, the head-mounted microphone commonly used turned out to provide excellent sound for human recognition as well. The remaining challenge was to get that sound from the microphone to the wizard.

Once the wizard heard the user’s command, he or she would type the appropriate code or error message and transmit that text to the user. To be believable, the text would have to appear all at once, not keystroke by keystroke as if being typed. Also, there should be no typographical errors—but this burden would rest on the wizard.

For the purposes of the study, we decided to run speech recognition software during the session. If the wizard could see the speech recognition output, it might serve as an aid to memory for long utterances, or even to help figure out what was said if the wizard didn’t recognize it.

More important than helping the wizard, we also wanted to record the speech recognition output. Seeing how close it came to being correct with this kind of usage would help us determine what to expect from a natural language programming system entirely implemented in software.
The speech software we use—Dragon NaturallySpeaking\(^1\)—takes sound directly from the sound card. Therefore it would have to run on the user machine and we would have to get its text output back to the wizard. Also, NaturallySpeaking is Microsoft Windows software, so the user machine would have to run that operating system.

Not only did we want to transmit sound from the wizard to the user, and text both directions, but we wanted to keep a synchronized record of all this correspondence for later analysis. To facilitate reviewing hour-long sessions, the data would have to be broken into smaller, ordered fragments. We wanted to be able to quickly access the audio of a user issuing a command, together with the speech recognition output and the wizard’s response.

Finally, since our test users were to be drawn from an introductory programming class and were to use our system to work on their homework assignments, we needed a way to get the files to them after each session.

### 4 The Audio: Network Streaming, Playback, and Segmentation

Handling the audio turned out to be somewhat more involved than it seemed at first. Early on, we considered using software designed for conducting meetings over a computer network. Such software could not only transmit sound from the user to the wizard, we thought, but might also be used to communicate text. Unfortunately, it didn’t seem flexible enough to create a convincing illusion. Keystrokes would have been sent one by one, and the original way we envisioned using it would have required the wizard to move the user’s mouse cursor, a dead giveaway that more than software is involved.

Next we tried using software designed for streaming MP3 audio over the Internet. However, this software buffers a certain amount of the data to prevent the sound from cutting out due to network glitches. This buffering created too much latency in the audio stream, exacerbating the already long delay between command and response. We might have been able to modify the open source audio streaming server icecast\(^2\) to use a smaller buffer, but we found a better solution first.

Justin Frankel wrote for us a program called pcmserv which listens for a single

\(^1\)http://www.dragonsys.com/
\(^2\)http://www.icecast.org/
connection on a command-line specified TCP port. When it gets a connection, it starts sending monaural, 16 bit, 22.050kHz PCM data from the default Windows sound recording device. It stops when it is killed or when the connection is closed.

This elegant solution allows us a great deal of flexibility. The entire operation on the wizard machine consists of a single UNIX shell pipeline. Netcat\(^3\) gets data from pcmserv over the network and writes it to standard output. A program I wrote for this purpose, called pcmtee, reads the PCM data from netcat, converts it to Solaris byte ordering, and writes it to the Solaris audio device and to standard output. Finally, the NotLame MP3 Encoder\(^4\) reads the byte-swapped PCM data from pcmtee and writes a 32kbps MP3 audio file to disk.

To segment the audio, the pipeline is killed and restarted with a different output filename each time the wizard updates the code.

## 5 Presenting the Output: The User’s Interface

The presentation to the user of the wizard’s responses is the most critical aspect of the system with regard to the illusion we were trying to create, as it is the only feedback the user gets. Therefore we put much effort into making it appear an ordinary program interface.

To present the wizard’s updates to the code as atomic changes rather than keystroke by keystroke, we came up with the concept of a “viewer”. This interface would look to the user something like a text editor with an ordinary buffer, but it would in fact merely display the contents of a file on a network file system. The wizard, whose machine would also have access to this file system, would simply edit the code with a text editor. Each time the wizard finished translating a user’s command into code, he or she would save the file, causing the viewer on the user’s screen to update.

Since we wanted the user to be able to work with multiple files, and since we wanted the interface to include the notions of text selection and cursor position, the viewer needed to be slightly more complicated. Thus the viewer reads an “infofile” which contains the name of the file to be displayed and locations for the selection and cursor.

To know when to reread the infofile, the viewer periodically polls for a “lock-file”. When a file with this specific name appears, the viewer deletes the lockfile,

\(^3\)http://www.10pht.com/~weld/netcat/
\(^4\)http://hive.me.gu.edu.au/not_lame/
Figure 1: The user’s interface. Above is the code window, below are messages. The prompt is in the grey bar between.
reads the infofile (which is also in a specific location), and updates its display accordingly.

Initially we had two separate viewers; one for the code and another for messages when code could not be generated. Later we decided to combine these into one window, which is more conventional from a user interface perspective, easier to use and more believably a software application. But behind the scenes, there are two separate infofiles, two separate lockfiles, and so on.

6 Handling the Speech Recognition Output

The viewer is also instrumental in getting speech recognition output to the wizard. The user-side viewer accepts text input from NaturallySpeaking, which simply sends keypress events to the window with focus. Each time the viewer intercepts a carriage return, it appends the new text to the end of a file on the network file system, and writes a corresponding infofile and lockfile.

Because the end of a new line signifies a completed command, we tell the users to say “new paragraph” when they’re done stating their requests and ready for them to be processed. Saying “new line” would accomplish the same thing, but since a line might refer to a line of code, we think the “new paragraph” command is less prone to confusion.

To make the system seem more application-like, saying “new paragraph” causes a message in the window to change from “Please state your next request, followed by ‘new paragraph’…” to “Processing your request. Please wait.” When the wizard updates the user’s display, the message changes back.

The wizard’s view of the speech recognition output is another instance of the old single-file viewer, and works the same way as the viewer on the user machine. Another reasonable way to accomplish the same thing is to run the UNIX command `tail -f <speech file>`, which follows the end of the file, displaying text as it is appended.

There is one complication with running NaturallySpeaking on the user machine. Since Windows provides no mechanism for two programs to read from an audio device at the same time, we had to install two sound cards in the user machine. Luckily, NaturallySpeaking can be configured to use a specific device, rather than just using the Windows default device. This way we are able to prevent it from conflicting with pcmserv.
Figure 2: The wizard’s interface. To the left are the emacs windows, to the right the speech recognition output and the audiocontrol window.

7 Running the Show: The Wizard’s Interface

Besides the speech recognition output viewer, the wizard’s interface consists of a pair of Emacs editors: one for code, and the other for messages. Each instance of Emacs is customized with its own Emacs Lisp to perform several key tasks.

Central to the Emacs Lisp is the update function, which is activated when the wizard presses Control-U. In the message editor, it writes the contents of the buffer to a file, then generates an infofile and a lockfile, and displays a confirmation message in the Emacs minibuffer. If the buffer hasn’t been modified since the last update, it skips writing the files and displays a message to this effect.

When the code editor’s update function is invoked, it first runs a simple check: if there is no new text in the speech recognition output file, it displays a message
to that effect and terminates. This prevents the wizard from accidentally updating the user’s display when the user has not finished issuing a command. For example, it is all too easy for a human wizard to recognize the user is verbally finished with a command, even if the user hasn’t said “new paragraph”. Updating the display in this situation might give away the game.

If there is new speech recognition output when update function is invoked, it makes a record of additions to the message file and the speech recognition output file since the last update. It does this by running diff against the previous versions of the files, redirecting the output to file names containing the code filename and revision number. (This naming scheme makes it simpler to correlate revisions of the files later.) Once diff is done, the current versions of these two files are copied over the old ones to prepare for the next revision.

If there is an audio pipeline running, it is killed; in any case a new one is started. The MP3 output filename, like the others, contains the code file name and revision number. Killing and restarting the pipeline causes a short blip in the playback, but it hasn’t been a problem, since by design this should happen some time after the user has finished issuing a command.

Finally, the function creates an infofile and a lockfile to signal the user’s viewer, and it displays a message in the Emacs minibuffer to indicate it is done. In addition to simply updating the user’s viewer with the wizard’s output, we later decided it would be useful to have a few more features in the wizard’s interface. For example, we made it possible to compiler the code and send any compiler messages to the user. To do this, the wizard uses Emacs’s Meta-X compile in the message window.

Another thing we realized is that the audio at the beginning of a session is usually a long period of silence before the user enters the room, gets situated, and starts issuing commands. To keep from recording all this, the wizard can press Control-R in the code editor to restart the audio pipeline restarted with the same output filename. Anything just recorded is clobbered.

8 Sending Files to the User

Since our users use our system to do their homework, they need to get copies of the files they work on during the session. We accomplish this with a Perl script called sendfiles.pl. The script collects all the files in the directory where code is stored during the session, and constructs an e-mail with a From: header indicating the pseudo-user for our project, nlprog@cs.utah.edu.
The files are encoded as MIME 1.0 attachments, which is as simple as inserting a few boundary lines between the concatenated files, because they are plain text and can be included with no modification. The most complicated thing about sendfiles.pl is the way it gets the e-mail address to use.

Since we were required to make some effort to obscure the identities of our human test subjects, we decided to only store their e-mail addresses in an encrypted file, misc/users. This file contains a Perl hash—plain text code—correlating our anonymizing usernames with the users’ e-mail addresses. It is encrypted using UNIX’s crypt command, with the nlprogs user’s password as the key.

When sendfiles.pl is given a command-line parameter, it interprets it as one of our anonymizing usernames. It calls crypt to decipher the user file; crypt prompts for a key. If a wrong key is entered, sendfiles.pl tries again, ad infinitum. When called without a username, or with one that is not in the user file, sendfiles.pl prompts for an e-mail address to which to send files.

To update the user file, it is necessary to decrypt it, change the Perl code, and encrypt the new code. Since crypt implements a symmetric cipher, both command lines take the same form: crypt < file1 > file2. Naturally, the key needs to be entered each time. It is important to verify the correct key was used to encrypt the Perl code before deleting the unencrypted file; if it gets encrypted with a mistyped key, the data may be lost.

9 Collating the Data

After the session is over, the various files must be put together into a coherent structure for review and analysis. This, too, is accomplished with Perl.

The script collate.pl begins by listing the files in the audio directory in order by modification time. This list is, in effect, a chronological sequence of completed user commands during the session. Since the file naming convention is the same for the speech recognition output and message files, a listing of either of those could equally well be used.

The script generates an HTML document with a series of tables; each table represents a completed command and is labeled with the code filename and RCS revision number to which it corresponds. The left column of the table contains indicators—audio, transcript, messages, and code—which describe the content of the cells to their right.

For the audio, there is a hyperlink to the appropriate MP3 file. The transcript from NaturallySpeaking and any messages sent to the user are included verbatim.
For the code, collate.pl uses rcsdiff to show what the wizard changed in the file since the previous revision.

With such a document, reviewing sessions is less work than it otherwise might be. One can scan through the revisions, or search the text for one command in particular, and have a quick summary at the fingertips. With a Web browser configured to launch an MP3 player, one can hear the user’s verbal command with the activation of a single link.

To function properly, collate.pl must be run in a session directory—one with subdirectories called audio, code, msg, and so on. The HTML document is written to standard output; the output may be redirected to a file if desired.

10 Running a Session

To simplify running user sessions, we wrote scripts to start the necessary programs on both the user machine and the wizard machine. NaturallySpeaking needs to be trained on a person’s voice before it can recognize that person’s speech, so this must be done ahead of time.

Before the user arrives for the session, the nlprog pseudo-user should be logged into the user machine. This user has read and write access to our directories on the network file system, which we don’t want other people browsing since we need to keep the nature of our system a secret.

The user start-up script, user.bat, starts NaturallySpeaking, pcmserv, and the viewer. The user’s profile must be selected manually from the NaturallySpeaking dialog. When NaturallySpeaking is finished launching, it is important to make sure the microphone icon in the system tray indicates NaturallySpeaking is ready to recognize speech. The actual microphone should be plugged into the machine and switched on.

The viewer application should be the only thing visible on the screen. The Windows task bar should be hidden and any desktop icons should be obscured, and the mouse cursor should be off-screen. At this point the wireless keyboard and mouse should be removed from the room so the user cannot snoop around in the system.

On the wizard machine, startsession.sh takes care of removing any old lockfiles from previous sessions, creating placeholder infofiles for the viewer, and starting the editors and speech recognition output viewer which comprise the wizard’s interface. In addition, it starts the Solaris audiocontrol program so the wizard can easily adjust the playback volume during the session.
If pcmser\text{v} is running on port 4444 of the user machine when the code editor is started, sound playback and recording starts immediately. The user machine’s hostname and the pcmtee port number are hard coded in one place in the \textit{wizard.el} script; the same port number is hard coded in \textit{user.bat}.

Once the editors are started, the wizard can edit files in the code editor; pressing \textbf{Control-U} invokes the update function. Likewise, typing into the message editor and pressing \textbf{Control-U} causes the message to appear in the user’s viewer, along with a beep and a brief flash.

When the session is over, all the applications should be closed and the script \textit{endsession.sh} should be run. This script takes one command-line parameter: a username. It prompts for whether to send files to this user, and if the response is yes, it calls \textit{sendfiles.pl} with this username. Next it moves the current session to a directory whose name is the current date, located in the directory whose name is the username. Finally, it creates a new empty directory structure for the next session.

The last thing to do is to run the data collation script in the newly archived session directory; but this need not be done immediately after the session.

\section*{11 Conclusions}

Because our system relies on writing to and reading from a network file system for all of its text communication, it is subject to any delays introduced in either phase of that operation. Changes in the state of the file system on the wizard machine are not instantaneously visible on the user machine, and vice-versa. Our experience is that most of the time the delay is acceptable, but that it is highly variable, perhaps because of network congestion or other factors external to our system. Direct communication over the network would perform more consistently and involve less latency on the whole, but we chose to use the file system because it was quicker to implement.

Another problem we had was with the user interface. Since the users’ only input was speech, they spent quite some time positioning the cursor and navigating the contents of the file. We didn’t make it possible for the wizard to scroll the user’s buffer, or even to see what was visible in the buffer; scrolling only happens as a result of cursor movement. This was a source of some frustration for our users.

Also, our wizards were—being human—somewhat less than perfect in typing code. Once or twice a user remarked something like, “It’s weird; it got it right
once but wrong another time.” We passed off such errors as bugs; after all, this is a cutting edge research program, still in development, right? Alan Turing addressed this issue in his 1950 paper: “If the man were to try and pretend to be the machine he would clearly make a very poor showing. He would be given away at once by slowness and inaccuracy in arithmetic.” One might reasonably extend this beyond arithmetic to nearly any task a computer can perform.

Despite these shortcomings, however, we’ve managed to keep our wizard behind the curtain to this point, as far as we know. None of the ten users have voiced any suspicions so far; some have even asked when the software will be offered for sale! We take this as a good sign, but we recognize more work is necessary to improve the user interface in particular; shortcomings in this area obscure the data we’re trying to collect.

12 Acknowledgments

This project was advised by Professors Joseph Zachary and Ellen Riloff, driven largely by graduate student David Price, and enabled by the contribution of my fellow undergraduate Ben Newton. The work described here was supported in part by the NSF under grant IIS-0090100.
References


A Appendix

A.1 wizard.el: Wizard Code Window (Emacs Lisp)

;; wizard.el
;; 2000.08.30
;; Ben Newton (bnnewton@cs.utah.edu) and Dana Dahlstrom (dana@cs.utah.edu)

;; Emacs lisp code for the NLP Naturally Java Tests.
;; Emacs must be started from the code directory for audio to work.

;;; Accessors for directory and file names, command lines, etc.

(defun tmp-mp3-file-name () "../audio/tmp.mp3")
(defun mp3-file-name () (concat "../audio/" (file-name-version) ".mp3"))

(defun tmp-msg-file-name () (concat "../msg/messages"))
(defun old-msg-file-name () (concat "../msg/messages.old"))
(defun msg-file-name () (concat "../msg/" (file-name-version) ".txt"))

(defun tmp-speech-file-name () (concat "../transcripts/speech.out"))
(defun old-speech-file-name () (concat "../transcripts/speech.out.old"))
(defun speech-file-name () (concat "../transcripts/" (file-name-version) ".txt"))

(defun file-name-version ()
  (concat (file-name-nondirectory buffer-file-name) 
  "-" 
  (vc-workfile-version buffer-file-name)))

(defun recording-command ()
  (concat 
  "/res/nlp/nlprog/bin/nc rivendell 4444 |" 
  "/res/nlp/nlprog/bin/pcmtee -x |" 
  "/res/nlp/nlprog/bin/lame -h -s 22 -b 32 -m m --voice - " 
  (tmp-mp3-file-name))))

;; End the previous recording process if there is one, but don’t save the file.
(defun just-record ()
  (interactive)
(when (get-process "recording")
  (delete-process "recording")
  (start-audio-recording)
)

; This is a function which saves the file the wizard is working on,
; and also outputs a file with information on selection, cursor position
; filename, and what line the cursor is on. This information is used by the
; Code viewer to display a copy of what the wizard creates.
; As soon as update is called (with the C-u key combination) the Code Viewer
; updated to match what the wizard sees.
(defun update ()
  (interactive)
  (if (= 0 (call-process "diff" nil nil nil
                 "-h" (old-speech-file-name) (tmp-speech-file-name)))
    (message "No new speech output. NEW PARAGRAPH!")
    (progn
      (update-code-file)
      (new-msg-revision)
      (new-speech-revision)
      (end-audio-recording)
      (start-audio-recording)
      (make-info-file)
      (message "File saved, version updated, infofile written.")))
)

(defun start-audio-recording ()
  (start-process-shell-command "recording" nil (recording-command)))

(defun end-audio-recording ()
  (when (get-process "recording")
    (delete-process "recording")
    (rename-file (tmp-mp3-file-name) (mp3-file-name)))

(defun update-code-file ()
  (set-buffer-modified-p t)
  (save-buffer)
  (new-revision buffer-file-name))

(defun new-msg-revision ()
  (shell-command-to-string
(concat "diff " (old-msg-file-name) " " (tmp-msg-file-name)
 "> " (msg-file-name)))
(shell-command-to-string
 (concat "cp " (tmp-msg-file-name) " " (old-msg-file-name))))

(defun new-speech-revision ()
 (shell-command-to-string
 (concat "diff " (old-speech-file-name) " " (tmp-speech-file-name)
 "> " (speech-file-name)))
(shell-command-to-string
 (concat "cp " (tmp-speech-file-name) " " (old-speech-file-name))))

(defun new-revision (file-name)
 (vc-next-action-on-file file-name nil "") ; check in
 (vc-next-action-on-file file-name nil "") ; check out

(defun make-info-file ()
 (let* ((fname (buffer-file-name))
        (smallfname (file-name-nondirectory fname))
        (pt (point))
        (ln (what-line))
        (rb (if mark-active (region-beginning) -1))
        (re (if mark-active (region-end) -1)))

 (generate-new-buffer "buff")
 (set-buffer "buff")

 ; print info to the buffer
 (princ (with-output-to-string
         (princ fname) (terpri)
         (princ smallfname) (terpri)
         (princ pt) (terpri)
         (princ rb) (terpri)
         (princ re) (terpri)
         (princ ln))
 (get-buffer "buff"))

 (write-region (point-min)
               (point-max)
               "/res/nlp/nlprog/view/infofile"

18
nil)
    ; save the buffer to a file
    (write-region (point-min)
    (point-min)
    "/res/nlp/nlprog/view/lock"
    nil)
    ; save the lock file
    (kill-buffer "buff"))

;;; Stuff that happens at load time.

(global-set-key \C-u 'update)
(global-set-key \C-r 'just-record)

(setq make-backup-files nil)
(setq transient-mark-mode t) ; set the buffers to use transient mark mode
(setq vc-checkin-switches "-f") ; force new RCS revision even if no changes

; (load-file "/usr/local/share/emacs/20.4/lisp/vc.el") ; eng side
(load-file "/usr/local/gnu/share/emacs/20.4/lisp/vc.el")

; (find-file "Scratch.java"); when using java
(find-file "scratch.cpp"); for cpp mode
(start-audio-recording)

A.2  msg.el: Wizard Message Window (Emacs Lisp)

; msg.el
; 2001.08.30
; Ben Newton (bnewton@cs.utah.edu) and Dana Dahlstrom (dana@cs.utah.edu)

; Emacs lisp code for the NLP Naturally Java Tests.
; Emacs must be started from the code directory for audio to work.

; This is a function which saves the file the wizard is working on,
; and also outputs a file with information on selection, cursor position
; filename, and what line the cursor is on. This information is used by the
; Code viewer to display an copy of what the Wizard creates.
; As soon as update is called (with the C-u key combination) the Code Viewer
(defun update ()
  (interactive)
  (if (buffer-modified-p)
      (progn
        (write-file "messages")
        (make-info-file)
        (message "File saved, infofile written.")
        (message "Buffer not modified. What’s wrong with you?")
      )
  )
)

(defun make-info-file ()
  (let* ((fname (buffer-file-name))
         (smallfname (file-name-nondirectory fname))
         (pt (point))
         (ln (what-line))
         (rb (if mark-active (region-beginning) -1))
         (re (if mark-active (region-end) -1)))
    (generate-new-buffer "buff")
    (set-buffer "buff")
    (princ fname) (terpri)
    (princ smallfname) (terpri)
    (princ pt) (terpri)
    (princ rb) (terpri)
    (princ re) (terpri)
    (princ ln))
  (get-buffer "buff")
)

(write-region (point-min)
  (point-max)
  "'/res/nlp/nlprog/view/msginfofile"
  nil)

; save the buffer to a file
(write-region (point-min) (point-min)
  "'/res/nlp/nlprog/view/msglock"
nil)
    ; save the lock file
    (kill-buffer "buff"))

(defun javac () "/res/nlp/nlprog/Solaris_JDK_1.2.2_07/bin/javac") ;for java
(defun classfiles-dir () "/res/nlp/nlprog/sessions/current/classfiles") ;for java

(defun compile ()
  (interactive)
  (cd "../code")
  (let* ((firstline (shell-command-to-string
                      "/bin/head -1 /res/nlp/nlprog/view/infofile"))
          (path (substring firstline 0 (1- (length firstline))))
          (filename (file-name-nondirectory path)))
    (insert (concat "--- Compiling " filename "\n"))
    (call-process "g++" nil t nil "-S" "-o" "/dev/null" filename)
    ; (call-process (javac) nil t nil "-d" (classfiles-dir) filename) ;for java
    (insert "--- Done.\n")
  (cd "../msg")

(defun message-cnu () (interactive) ; (beginning-of-line) (kill-line)
  (insert "Command not understood.\n"))

;;; Stuff that happens at load time.
(global-set-key "\C-u" 'update)
(global-set-key [f1] 'message-cnu)

(setq make-backup-files nil)
(setq transient-mark-mode t) ;set the buffers to use transient mark mode
;(load-file "/usr/local/share/emacs/20.4/lisp/vc.el") ;eng side
(load-file "/usr/local/gnu/share/emacs/20.4/lisp/vc.el")

(find-file "messages")
A.3  **user.bat: User Start Script (DOS Batch)**

REM user.bat
REM 2001.04.08
REM Dana Dahlstrom (dana@cs.utah.edu)

@echo off

start C:\NatSpeak\Program\natspeak.exe

start /min T:\nlp\nlprog\bin\pcmserv.exe 4444

T:
   cd \nlp\nlprog\view
   start /min C:\jdk1.3\bin\java.exe viewer

A.4  **startsession.sh: Wizard Start Script (Bourne Shell)**

#!/bin/sh

# startsession.sh
# 2001.05.23
# Dana Dahlstrom (dana@cs.utah.edu)

# script to set up things on the wizard computer for the NLP Natural
# Programming study and tests.

umask 007

nlprogdire=/res/nlp/nlprog
currentdir=$nlprogdire/sessions/current

# Remove old lock files.
for file in $nlprogdire/view/lock $nlprogdire/view/msglock ; do
   rm $file 2>/dev/null
done

# Put infofiles in place to make viewer happy.
cp $nlprogdire/view/startinginfofile $nlprogdire/view/infofile
   cp $nlprogdire/view/startingmsginfofile $nlprogdire/view/msginfofile
# So emacs uses sh.
SHELL=/bin/sh; export SHELL

# Start programs for wizard interface.
( cd $currentdir/code && emacs -l $nlprogdir/elisp/wizard.el \
-geometry 120x38+0+40 & )
( cd $currentdir/msg && emacs -l $nlprogdir/elisp/msg.el \
-geometry 120x17+0+707 & )
( cd $nlprogdir/speechview && ../Solaris_JDK_1.2.2_07/bin/java speechviewer & )
( audiocontrol -geometry +754+820 & )

A.5  endsession.sh: Wizard End Script (Bourne Shell)

#!/bin/sh

# endsession.sh
# 2001.05.23
# Dana Dahlstrom (dana@cs.utah.edu)

# script to get things ready for the next session. (ie copy files from
# current, etc)

if [ $# != 1 ]; then
    echo "Usage: $0 <username>"
    exit
fi

umask 007

user=$1
nlprogdir=/res/nlp/nlprog
currentdir=$nlprogdir/sessions/current
userdir=$nlprogdir/sessions/$user
date='date +%Y.%m.%d'

if [ ! -d $userdir ]; then
    echo "$userdir does not exist. Aborting."
    exit
fi
if [ -d $userdir/$date ]; then
    echo "$userdir/$date already exists. Aborting."
    exit
fi

echo -n "Send files to $user (yn) [n]? "; read send
if [ "$send" = "y" ]; then
    /res/nlp/nlprog/bin/sendfiles.pl $user
    if [ "$?" != "0" ]; then exit; fi
fi

echo "Deleting $currentdir/classfiles"
rm -rf $currentdir/classfiles

echo "Moving current session to $userdir/$date"
mv $currentdir $userdir/$date

echo "Creating empty session structure in $currentdir"
for subdir in audio code code/RCS msg transcripts classfiles; do
    mkdir -p $currentdir/$subdir
done

# Touch these files so the viewers don't choke without them.
for file in code/scratch.cpp msg/messages msg/messages.old \
    transcripts/speech.out transcripts/speech.out.old; do
    touch $currentdir/$file
done

# Truncate the error file
cat /dev/null > /res/nlp/nlprog/view/errors

A.6  sendfiles.pl: File Mailing Script (Perl)

#!/usr/local/bin/perl

# sendfiles.pl
# 2001.04.16
# Dana Dahlstrom (dana@cs.utah.edu)
if ($ARGV[0]) {
    %uhash = eval 'crypt < /res/nlp/nlprog/misc/users' until ($uhash{loaded});
}

$mailserver = 'mailhub.cs.utah.edu';
$hostname = 'hostname'; chomp $hostname;
$date = 'date +"%a, %e %b %Y %T -0600 (%Z)"'; chomp $date;
$sender = 'nlprog@cs.utah.edu';
$boundary .= ('.','/',0..9,'A'..'Z','a'..'z')[rand 64] foreach (1..16);
$body = 'cat /res/nlp/nlprog/misc/questionnaire.txt';
$codedir = '/res/nlp/nlprog/sessions/current/code';

unless ($recipient) {
    print "No such user in db: $ARGV[0]\n" if $ARGV[0];
    print "Enter e-mail address [blank to abort]: ";
    $recipient = <STDIN>; chomp $recipient;
    die "Aborted.\n" unless $recipient;
    $manual = 1;
}

print "Sending files to " . ($manual ? $recipient : $ARGV[0]) . "\n";

open MAIL, "|/res/nlp/nlprog/bin/nc $mailserver 25 > /dev/null"
    || die "Can’t connect to $mailserver!\n";
select MAIL;

print <<EOF;
HELO $hostname
MAIL FROM: $sender
RCPT TO: $recipient
DATA
Date: $date
From: NLP User <$sender>
To: $name <$recipient>
Subject: $subject
Mime-Version: 1.0
Content-Type: multipart/mixed; boundary="$boundary"

25
chdir $codedir;
opendir CODEDIR, $codedir;

while ($_ = readdir CODEDIR) {
  if ( -f $_ ) {
    print "--$boundary
    print "Content-Type: text/plain; charset=us-ascii
    print "Content-Disposition: attachment; filename="$_"
    print "\n"
    print 'cat $_';
    print "\n";
  }
}
print "--$boundary--\n"

print ".\n"
print "QUIT\n"

closedir CODEDIR;
close MAIL;

A.7  **collate.pl: Data Collation Script (Perl)**

#!/usr/local/bin/perl

# collate.pl
# 2001.04.08
# Dana Dahlstrom (dana@cs.utah.edu)

die "Must run in a session directory.\n" unless ( -d "audio" and -d "transcripts" and -d "msg" and -d "code" );
$session = 'basename \'pwd\''; chomp $session;
$user = 'basename \'dirname \"pwd\"\''; chomp $user;

print <<EOF;
<HTML>
<HEAD>
<TITLE>User $user, session $session</TITLE>
</HEAD>
<BODY>
<CENTER><H2>User $user, session $session</H2></CENTER>
EOF

foreach ('ls -rt audio') {
    chomp;
    if (!$rev) {
        $filename = $';
        $rev = substr $&', 3, length($&')-7;
        print "<HR>
<H3>$filename, revision $rev</H3>
" .
        "<A HREF="audio/$\"\"mp3</A>/\"\"mp3</A></TD></TR>
    print "<TR VALIGN=top><TD><STRONG>Transcript</STRONG></TD><TD>" .
        'cat transcripts/${filename}-1.${rev}.txt' . "</PRE></TD></TR></H3>
    print "<TR VALIGN=top><TD><STRONG>Messages</STRONG></TD><TD>" .
        'cat msg/${filename}-1.${rev}.txt' . "</PRE></TD></TR></H3>
    print "<TR VALIGN=top><TD><STRONG>Code</STRONG></TD><TD>" .
        &code . "</PRE></TD></TR></H3>
    print "</TABLE></H3>
}

print "</BODY></HTML>\n";

sub code {
    $prev = $rev - 1;
    if ($prev) {
        $contents = 'rcsdiff --quiet -r1.$prev -r1.$rev code/$filename';
    } else {
        $contents = 'co --quiet -p1.$rev code/$filename';
    }
}
A.8 pcmtee.c: PCM Tee Program (C)

/* pcmtee.c
 * 2001.05.08
 * Dana Dahlstrom (dana@cs.utah.edu)
 * Adapted from BSD tee.c
 */

#include <sys/audioio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef struct _list {
    struct _list *next;
    int fd;
    char *name;
} LIST;
LIST *head;

void add(int, char *);
static void usage(void);

int main(argc, argv)
    int argc;
    char *argv[];
{
register LIST *p;
register int n, fd, rval, wval;
register char *bp;
int append, ch, byteswap, exitval;
char *buf;
char *swapbuf;
#define BSIZE (8 * 1024)
append = 0; byteswap = 0;
while ((ch = getopt(argc, argv, "aix")) != -1)
    switch((char)ch) {
    case 'a':
        append = 1;
        break;
    case 'i':
        (void)signal(SIGINT, SIG_IGN);
        break;
    case 'x':
        byteswap = 1;
        break;
    case '?':
        default:
            usage();
            break;
    }
argv += optind;
argc -= optind;

if ((buf = malloc((u_int)BSIZE)) == NULL) {
    perror("malloc");
    exit(1);
}

if ((swapbuf = malloc((u_int)BSIZE)) == NULL) {
    perror("malloc");
    exit(1);
}

add(STDOUT_FILENO, "stdout");

if ((fd = audio_open()) < 0) {
exitval = 1;
} else {
    add(fd, "audio device");
}

while ((rval = read(STDIN_FILENO, buf, BSIZE)) > 0) {
    if (byteswap) {
        char* bp = buf;
        char* sbp = swapbuf+1;
        int copied = 0;

        memcpy(swapbuf, buf+1, BSIZE-1);

        while (copied < BSIZE) {
            *sbp = *bp;
            bp += 2; sbp += 2; copied += 2;
        }
        bp = buf; buf = swapbuf; swapbuf = bp;
    }

    for (p = head; p; p = p->next) {
        n = rval;
        bp = buf;
        do {
            if ((wval = write(p->fd, bp, n)) == -1) {
                fprintf(stderr, "%s", p->name);
                exitval = 1;
                break;
            }
            bp += wval;
        } while (n -= wval);
    }

    if (rval < 0) {
        perror("read");
        exit(1);
    }

    exit(exitval);
static void usage() {
    (void)fprintf(stderr, "usage: audiotee [-ai]\n");
    exit(1);
}

void add(fd, name)
    int fd;
    char *name;
{
    LIST *p;

    if ((p = malloc((u_int)sizeof(LIST))) == NULL) {
        perror("malloc");
        exit(1);
    }
    p->fd = fd;
    p->name = name;
    p->next = head;
    head = p;
}

int audio_open()
{
    audio_info_t ainfo;
    audio_device_t ad;
    char device[PATH_MAX];
    int fn;

    if (getenv("AUDIODEV"))
        strcpy(device, getenv("AUDIODEV"));
    else
        strcpy(device, "/dev/audio");

    if ((fn = open(device, O_WRONLY)) < 0) {
perror(device);
return fn;
}

if (ioctl(fn, AUDIO_GETDEV, &ad) == -1) {
perror(device);
return -1;
}

AUDIO_INITINFO(&ainfo);

ainfo.play.encoding = AUDIO_ENCODING_LINEAR;
ainfo.play.precision = 16;
ainfo.play.sample_rate = 22050;
ainfo.play.channels = 1;

if (ioctl(fn, AUDIO_SETINFO, &ainfo) == -1)
  return -1;

return fn;