The Pennsylvania State University
The Graduate School
Capital College

An Emotion Interface for Eliza

A Master’s Paper in
Computer Science
by
Robert G. Eyer

©2001 Robert G. Eyer

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

October 2001
Abstract

The focus of this project is threefold. First, this project demonstrates that wrapping the basic logic with an additional layer that emulates emotion may incrementally enhance the AI capabilities and functionality of an MIT ELIZA program. This feature provides mood states that can be triggered by user input. This new ‘Emotion Interface’ is initially set to implement the emotions of ‘anger’ and ‘reflection’. Second, the modular addition of video images and audio capabilities further enhance Eliza, providing output, which includes both verbal and non-verbal communication. And third, the project provides a toolkit whereby a user can fully modify and extend Eliza scripts, allowing script development by non-programmers.

The Eliza program, originally developed by Joseph Weisenbaum is essentially a parser program that looks at a user input sentence, identifies keywords, and constructs what is hopefully an appropriate response using some clever linguistic tricks.

The re-implementation of Eliza is in Java, and includes a 3D graphic talking head with lip sync speech capabilities, controllable expressions, and lighting control of the graphic background. The Script Editor toolkit allows the user to make script changes on all basic Eliza script elements as well as the newly added Emotion Interface.
# Table of Contents

Abstract i

Acknowledgement iv

List of Figures v

1 Introduction 1

2 History of Eliza 2

3 AI Considerations 5

4 Language Modelling 8

5 Project Model 10

6 Eliza’s Parsing Logic 13
   6.1 Tokenizing .................................................. 13
   6.2 Pre-Transforms ............................................ 14
   6.3 Keywords .................................................. 14
   6.4 Decomposition Rules ..................................... 15
   6.5 Memory Flag .............................................. 15
   6.6 Input String Indices ..................................... 16
   6.7 Synonym Lists ............................................ 18
   6.8 Post-Transformations ..................................... 19
   6.9 Emotion Triggers ........................................ 20

7 Eliza Implementation 21
   7.1 Scripts ...................................................... 21
   7.2 File System ............................................... 21
   7.3 Emotion Layer ............................................ 22
   7.4 Display .................................................... 22
Acknowledgement

I wish to express my thanks to Ben Slotznick of Simtalk Inc., for sponsorship of this project. I would also like to thank IBM and the Haptek Corp. for use of their products during the development of this project.

The graphic plug-in and speech engine used in conjunction with this project is an activeX control produced by the Haptek Corporation. The plug-in module was downloaded from their web site at www.haptek.com.

The Java wrappers for the Haptek activeX control were produced with the help of an alpha release of an IBM product called Bridge2Java. This product is available for download from IBM’s alpha release web site at www.alphaworks.ibm.com.

And finally, I would like to thank the members of Penn State University’s Master’s Committee, Dr. Thang Bui, Dr. Linda Null, Dr. Pavel Naumov, and Dr. Sandra Kingan for their input and guidance.
List of Figures

1  Reassembly of the indices ........................................ 17
2  Script Object Diagram ........................................... 21
3  Program Data Flow ................................................. 23
4  Mood Start Up Algorithm .......................................... 24
5  Logic for Mood Input ............................................... 25
6  Program Flow ........................................................ 25
7  Parser Flow .......................................................... 26
8  Early version of the Eliza applet ................................. 27
9  Console Application .................................................. 28
1 Introduction

This project involves the design and implementation of an emotion interface for the basic Eliza program, and development of a toolkit which allows a non-programmer to create and modify Eliza script files which use that emotion interface. Further, by integrating a third party graphic/speech package, I was able to provide a more natural output to program users and implement both verbal and non-verbal responses to user input.

My own contributions to this project include the complete design and implementation of the Eliza Script Editor and all control code between the Java application and the Haptek Virtual Friend software. The script element classes are Java representations of the original Eliza script elements. All basis parser logic is true to Weisenbaum’s original Eliza program. The mood wrappers and all associated logic is of my own design, as are all file IO classes. The wrapper classes between the Java application and the Virtual Friend ActiveX control were generated using an alpha release of IBM’s Bridge2Java software. All documentation and graphics are of my own creation.

While the underlying parsing ability of Eliza is unchanged and insufficient for any real natural language recognition, this project demonstrates modularity, and layering of personality as a useful approach. Additionally it provides a tool by which linguists or hobbyists are able to experiment with scripting of the Eliza program. An additional accomplishment involves integration of a non-standard activeX control into a Java console application. This is a newly emerging technology and may be useful to others wishing to combine activeX and Java in a stand-alone application.

As project sponsor, Simtalk Inc. imposed additional project parameters as follows:

1. The project should be implemented in Java to insure ultimate portability to multiple platforms, but should initially focus on a stand-alone application for Windows 9X or NT.

2. The project should provide a GUI that implements the Haptek ActiveX ‘Talking Heads’ as a method of returning program output to users.
3. The ‘Emotion Interface’ and ‘Script Toolkit’ should be integrated into the basic GUI by use of a file system, eliminating the need for server involvement beyond distribution.

Specific project sub problems include:

- Design and implementation of a mood layer to wrap the existing Eliza parsing logic.
- Design a toolkit which is GUI driven that allows complete modification of Eliza scripts.
- Integration of scripting ability, (file writing), without the use of server or client databases.
- Integration of a 3rd party activeX control into a Java console application.

2 History of Eliza

Joseph Weizenbaum, an MIT Professor, won national acclaim and a Loebner prize for a program he developed in 1966 that simulated a Rogerian psychotherapist.

“... Named to emphasize that its language abilities may be incrementally improved by its users, since its language abilities may be continually improved by a “teacher”. Like the Eliza of Pygmalion fame, it can be made to appear more civilized, the relation of appearance to reality, however, remaining in the domain of the playwright.”[4]

The program parsed sentences input by a user, extracted keywords and other useful structural information, and finally gave a response, hopefully appropriate, based on those parsed keywords. While this program is a poor beginning, in terms of AI, of a computer attempting to understand natural
language, it must still be recognized as a beginning. Eliza does not seek to understand what the user is trying to communicate, but instead parses the users input, and then through a series of linguistic tricks, tries to return some minimally appropriate response, creating an impression that ‘Eliza’ understands and is capable of two way conversation. Legends persist about users who were fooled into thinking that they were speaking to a real psychiatrist while using the ‘Eliza’ program. However, as users become more sophisticated and knowledgeable of the existence of various Eliza like programs, known as ‘BOTs’ and ‘Agents’, it becomes increasingly difficult to fool a user for more than a few minutes.

Since 1966, large numbers of improvements have been made to the basic Eliza program, producing BOTs that are capable of performing useful functions. When tied to a knowledge base, these ‘Agents’ can become useful in answering user queries about a myriad of technical subjects. The Agents parse the user’s questions, and again, based on keywords and key phrases, guess the most appropriate entries of the knowledge base to return. I use the term guess in spite of the fact that the answers given are based on sophisticated algorithms which parse and search target libraries by weight. These Agents usually return a selection of possibly appropriate choices, allowing the user to select the most suitable or to try a different query. However, true AI machine ability, to parse and understand natural language, or simulate normal conversation is still an elusive goal. Since 1990, an annual contest, known as the Loebner competition, is held to determine which contestant program most closely simulates a real person. Based on a test proposed by Alan Turing in 19502, the Loebner competition offers an overall prize of $100,000 to any contestant who enters a computer program that can fool a panel of judges into believing it is a real person. The contestants are separated from the judges, but are connected to the panel by means of a computer terminal. The judge may be connected to a real person, or to a computer simulation. Communicating over the terminals, each judge must decide whether they are communicating with a real people, or a computer program. No one has yet fooled the judges, and won the grand prize. However, one contestant is an-
nually awarded a prize for the program, which most closely simulates a real person. One recent article, written by a contestant of the 1995 Loebner competition illustrates the difficulties of creating a program capable of fooling a judge. Speaking about his loss in this competition, Thom Whalen, the 1994 Loebner prizewinner states:

My error was that the judges, under Loebner's rules, did not treat the competitors as though they were strangers. Rather, they specifically tested the program with unusual questions like, "What did you have for dinner last night?" or "What was Lincoln's first name?" Weintrab's program, however, was a master of the non-sequester. It would continually reply with some wildly irrelevant statement, but throw in a qualifying clause or sentence that used a noun or verb phrase from the judge's question in order to try to establish a thin veneer of relevance.

I am amazed at how cheerfully the judges tolerated that kind of behavior. I can only conclude that people do not require that their conversational partners be consistent or even reasonable. But I am not ready to draw any conclusion about whether this is a fundamental problem with the Turing test.

Remember that we are talking about conversational partners that are fairly quickly recognizable as computer programs. To appear completely human, I would expect (hope) that the program would have to be much more responsive to the questions that were asked.[3]

After nearly thirty years of development, the BOTs, which are judged most human, must still depend on the same type of linguistic tricks employed by the original Eliza program. Even though modern parsing engines are faster and more efficient, and script sizes, (keywords and response rules), have grown by two or more orders of magnitude, very little improvement has been demonstrated in the areas of natural language recognition or machine personality simulation.
3  AI CONSIDERATIONS

Alan Turing, noted mathematician and computer scientist, prophetically stated that by the end of the century, (2000), that no one would be shocked by the idea of a computer that is able to communicate with a person. He also suggested that we would need internal storage in the computer sufficient to store one billion binary digits. While we have surpassed Turing's expectations in computer hardware, we still have not reached the point of computer comprehension of natural language.

3  AI Considerations

I would suggest that all of the hardware, algorithms, and language resources are already available to accomplish Turing's vision, but that current research and industry resources mainly focus in the area of expert systems where expectation of a monetary return closely follows commitment of time and resources.

I would further propose that, just as the human mind can be symbolically represented as a hierarchy of processes, any successful model of intelligence must similarly model those same processes. This problem is exceedingly complex, but can be addressed by a modular approach, where individual mental sub processes are simulated, and then linked together. So long as we attempt to shortcut the process by eliminating processes that are a normal part of our own recognition system, any simulation is doomed to failure.

Consider for a moment, all of the things that a person brings to any conversation. Start with a set of language skills. These would include a base vocabulary, including spelling and variant spelling of many words, rules of grammar, (both formal and colloquial), and semantics, (conceptualization of a group of words). This last idea, conceptualization, is perhaps the most important. True machine intelligence requires the ability to communicate on a conceptual level. Many common phrases have meanings unrelated to the component words. Take the phrase "down the road". In the context of one conversation it may be literal, meaning some distance away following a certain road, or it can mean "at a later time". It follows then that the dic-
tionary employed by any truly intelligent agent will need to first search for phrases, (concepts), represented by word chains, before addressing the individual words contained in the chain. Disambiguating rules would need to be provided where multiple concepts could be implied from the same word chain. Likewise, a dictionary would need to contain large numbers of names, both of people and of places. Hence, some system of linkage would be requires in any dictionary, providing links to pertinent information. If the name ‘Robert’ is encountered, an intelligent agent should know that this is a surname, and associate the nicknames Rob and Bob to it. If the name Benjamin Franklin is part of a conversation, it should be recognized as a ‘famous’ name, and linkage should point to an information structure about Franklin. Placing all of this information within the BOTs grasp is not such an extraordinary task, since most of the resources are already available in digital form only the linkage and logical search packages need to be constructed. Along with this linguistic skills package, a considerable amount of memory is needed for the actual conversation. Just as a person remembers what they recently said, and what was said to them, any intelligent agent would need to have this same ability to refer back to previous parts of the conversation. Just like a normal person’s short-term memory, an intelligent agent needs to remember, for a ‘reasonable’ time, what they said and what was previously said to them. This would require storage of the actual text of conversation, and classification of the importance of the concepts expressed, (how memorable). Additionally, there would need to be a normal decay function that allows the Agent to forget unimportant information when some reasonable time period has passed. This type of short-term memory is essential to the ability to communicate intelligently. In normal conversation, we often refer back to some subject raised earlier in the conversation. Eliza already has a rudimentary memory, but it is only used as a linguistic trick, to say something seemingly relevant when the parser finds no keyword matches in any current statement.

Just as every real person has his or her own personal history, and an emotional package, every intelligent agent could also be endowed with one. Each Agent could be assigned a personal history. Since most details of daily
living are readily forgotten within a few days, a comprehensive virtual history need only include important details from the distant past, and detailed daily activities for the last week. Does anyone remember what they had for dinner 10 days ago? Not I! However, I do remember what I ate today and yesterday. So why would a BOT need to remember what an average person normally forgets? My point is that these history parameters are not excessively large, and could even be machine generated and updated to provide a continuous personal history. Let the history generator start with early childhood, assign random common names the BOT's teachers, and randomly determine whether those teachers had an impact on you. If a low impact, they are forgotten as the program progresses, but if they had a very high impact, the BOT will remember that teacher, the subject, and perhaps a memorable experience, even when its personality is mature. Rather than try to hard code a history script, I suggest an evolutionary program starts at early childhood, randomly takes the BOT through all of the things common to and memorable about childhood. If the BOT were to simulate a person of age 50, the major world and local events of the decades of the sixties and seventies would need to be entered into the BOT memories. Any BOT simulating a fifty year old would need to remember the Kennedy assassination, and have some emotion assigned to the memory. By running a BOT through a training program as just described, it would acquire a personal history in much the same fashion as a real person. Training should continue by regularly updating BOT memory with weekly news summaries that can be retained or forgotten according to the importance assigned by the BOT's emotional package. Likewise a detailed daily history program would include meals, weather, and other trivial information that is remembered for only a short time. This type of programming would allow the BOT to enter a conversation with the same type of personal context that a real person has.

This list of programming requirements is certainly not complete, but it is representative of the complexity of the problem. However, most of these problems can be developed as modules and integrated to form a useful agent. Creation of a useful and believable machine personality is certainly a vast
project and one that will need many talents beyond programming skills.

4 Language Modelling

A lot of hard working and well intentioned AI programmers have taken the approach that if they can just come up with the magic, super parsing, algorithm, everything will start to work. The beguiling part is that, with a good algorithm, you can often produce impressive effects with very little code and with computer hardware. The problem is that eventually you hit a brick wall, and progress becomes difficult if not impossible. Good algorithms simplify the individual tasks, but the overall number of tasks involved in a good AI implementation is will eventually define the need for extremely large programs and information databases.

Numerous problems face anyone interested in the development of natural language recognition and machine personality. Like many mathematically related problems, the actual definition of the problem is elusive. At the uppermost level, we can state the problem in terms of the Turing Test that is the basis of the annual Loebner competition. That is, a program simulating a human personality should be indistinguishable from a human when a judging party interacts with it; so for so good. The next problem is to describe the process by which we determine what is rational conversation. Which can be divided into the sub-problems of how do humans communicate thoughts, and how do humans understand another human’s communications. Now this is the sticky part of the problem. We do not yet fully understand the thought process, let alone the processes by which we communicate our thoughts. Turing’s test is a valid means of evaluating a solution, but it is not a good starting point to actually solve the problem. A top-down approach, (from the most complex definition to the underlying sub-problems), to solve natural language problems is only useful when all of the underlying mechanisms are full understood. A more useful approach might be to work bottom-up, from the most basic principles of perception and language interpretation, adding layer upon layer of complexity to simulate more complex interaction,
and testing each new layer to determine whether it actually provides more complex functionality.

I believe that current research into very high level parsing of verbs, using FSAs will produce the groundwork for a new generation of intelligent agents. Noam Chomsky is quoted as saying that he did not believe that finite state automata could properly parse natural language because of the problem with nested ‘if’ statements. In normal conversation this is not a real issue. Granted, for any automata that can be created, it is probably possible to construct a sentence that will cause ambiguity problems, but by placing a small finite limit on possible nesting, the problem disappears. Since this type of deep nesting is not normal conversation, the problem exists purely as a theoretical consideration. If individual verbs can be parsed for case and modality, the stage is set for construction of broader and more appropriate categories of response[2].

English is a notably difficult language for parsers, simply because it has roots in so many other languages, we end up with a lot of situations where phrases are both common and ambiguous, requiring a determination of meaning from some disambiguating rule. For example, the phrase “John was a loser” could refer to the outcome of a card game, mentioned five sentences earlier in the conversation, or it could be a comment on the general state of some important issue of John’s personality. Part of the problem is the English language. Perhaps part of the solution will be to use a less ambiguous language, like Spanish, or Greek or even create a new machine translation language. Such a language would need simple and unambiguous to aid in machine parsing, translating and communication, yet rich enough to handle most ideas and conversational issues.

The current state of the art would appear to be the Cyc project. Hidden for nearly 15 years and financed primarily by Intel, Microsoft, and the DOD, Cyc is purported to utilize millions of parsing rules to parse natural
language strings into concepts. Public awareness of Cyc has slowly been building in recent months, since Cycorp, the Austin, Texas developer of the Cyc knowledge base announced that a portion of its massive knowledge base would be made available to the public under a Limited General Public License. The newly released public software is now available on-line at http://sourceforge.net/projects/opencyc/[1].

5 Project Model

Since it is the aim of this project to develop an emotional interface for the basic Eliza program, it is first necessary to define a machine model of emotion. As such, I will use my own experience, (introspection), as a guide. My model makes a number of, hopefully reasonable, assumptions:

- A person normally operates in some emotional base state, where reactions and conversational alternatives usually fall within some normal, predictable behavior.

- Elevated states can arise from both internal and external stimuli.

  - Internal stimuli can be modeled by probabilities that some internal event has occurred which places Eliza in a state of elevated emotion at initialization, and will continue until some decay function produces normal recovery from the heightened state or until a user’s interaction triggers a change of emotional state.

  - User interaction can trigger state changes that can cause Eliza to enter into a heightened emotional state, or return to a normal state from a heightened state.

There are numerous actions that take place in normal conversation which could trigger a switch to a different emotional state, but which cannot reasonably be modeled in a text only exchange. Like the internal stimuli above, they might be modeled by another random variable, but I don’t see the point
in any further random additions. Gestures, voice inflections and tonality are normal triggers and could be added in a far more sophisticated version of this program, where user input is also vocal, utilizing a speech recognition software. Recent research with BOTs has produced agents that are capable of recognizing gestures from video input. Further improvements in speech recognition may produce programs that can recognize emotion triggers on the basis of tonality and inflection.

Within the limited text-to-text model, let’s first consider the emotion of anger. I find that I can be induced to an angry state by a multitude of conditions. My anger could result from my own actions, such as hitting my finger with a hammer while working. Although directed anger at myself doesn’t normally last long, the resulting angry state may linger for a considerable time period. Any type of pain or discomfort may result in anger. Since these are triggers that are not the direct result of communication, I will attempt to model them by a random variable. That is, at the inception of any communication period, Eliza’s emotional states can be determined by a random variable, which represents the range of possibilities of Eliza’s emotional state at any given moment in time. I will allow the interface to be programmed with emotion weights, which represent the probability that Eliza is in some heightened emotional state at the time of initialization. For example, the user might weight anger at .05, and reflectivity at .02. This would represent a personality, which could be expected to be angry 5% and reflective 2% of the time. To keep the model simple, I will generate a random number between 0.0 and 1.0. If the random number is less than or equal to .05, Eliza will enter an angry state. If the number is greater than .05 but less than or equal to .07, Eliza will be reflective. A different model might allow Eliza to enter combined states, which could be added using the same logical model with the addition of a separate random variable for each emotion, yet this would not significantly improve the demonstration. The two basic emotions that I have chosen for this demonstration might even be considered mutually exclusive, because human actions tend to be dominated by only one of these moods at any given time. I base this observation on my own personality, so for the sake
of argument, I will only state that Eliza is programmed according to my own beliefs about my own emotional states. Additionally, whenever Eliza is initialized into a heightened emotional state, a second random number between 0.0 and 1.0 will be generated, and multiplied by the mood’s excitation range. For instance, if the random variable was found to be .04, the mood would be set to angry and the dice rolled again to determine how long the anger will last. If the range of excitation for anger was 300 seconds, and the next random variable came out at .3, then a timer would be started on a separate thread, which would restore Eliza’s emotional state to the default after 90 seconds had elapsed. I realize that the decay rate of elevated modes will not be evenly distributed over time. I am not looking for an accurate estimate of decay times. Rather, I am trying to create a modular GUI that can easily be modified to allow different functions, and parameters to model a given personality. In practice, I will not even bother with actual timers but will instead use transactions. If an anger trigger is encountered and the program is setting anger response randomly, a random number of transactions will be determined for the program to remain in an angry state. After each user input, the transactions remaining to normal will decrement. This approach has the same overall effect as using timers without the additional overhead of instantiating and running timer threads.

Since mood can be altered quickly during the course of conversation, triggers have been added that can cause Eliza to be angry, or reflective when she is not already in those states. A separate set of triggers causes Eliza to revert to a normal state from one or all of the elevated states.

These mood shifts will require a rule set that determines how the program reacts to triggers, given the current ‘State’ of the program, and a number of ‘State’ variables.

The emotion interface will allow the user to set most of the emotional interface parameters within some reasonable range. Anger and reflectivity weights will be able to be specified by the user, so long as the total weights of the emotions do not exceed 1.0. Maximum response and decay times, transactions, will be from 0 to 5 for each emotion. Even the rules table will
allow the user to select the reaction to some combinations of current ‘mood state’ and triggers encountered in any newly parsed statement.

When the current state is elevated in any given emotion, and the parsed start of the next statement is a trigger for some emotion, the user will be given options to determine the programs response to every combination of states and triggers. Perhaps the most interesting options will be when conflicting emotional states are encountered. If, for example, the mood is reflective, and an anger trigger is encountered, the user can choose to revert to normal, reduce reflectivity by the value of the anger or merely ignore any other emotion until the existing state has decayed. Some of these combinations may be suitable to simulate a “crazy” Eliza.

By allowing the interface to be easily changeable, a given user can experiment with different rules and parameters in an attempt to model some predetermined behavior. At any time the user can choose to alter the personality settings by clicking on the ‘Script Utility’ button. The user will be presented with a main menu screen that allows the user to choose between various sub-screens that allow the user to change any item of the Script or numerous state variables representing Eliza’s environment parameters and rules. When finished with set-up, the Eliza program is re-initialized and restarted.

6 Eliza’s Parsing Logic

To understand the extensions to the parsing logic that have been added to provide the emotional triggers, it helps to first understand the parsing system utilized in a normal Eliza program.

6.1 Tokenizing

Parsing begins by decomposing the user input into understandable units. Specifically, user input is pulled from the statement one word at a time, starting with the leftmost word in the statement. Periods and commas are
6 ELIZA'S PARSING LOGIC

separated from the words preceding them by padding the text with spaces. All letters are converted to lower case, and any unwanted punctuation and symbols are removed without substitution.

The modified words, periods, and commas of the decomposed sentence are then placed into a linked list and compared individually to all [pre:] transformations, starting with the first word in any input sentence.

6.2 Pre-Transforms

The [pre:] transformation is a matched pair of words where the first word, whenever encountered, is to be replaced by the second word in the pair. Thus, this search replaces any matched word with one that will be more easily recognized by later stages of parsing. This is one way to handle contractions, variant spellings, abbreviations, or even misspellings. For example, the script file entry, "pre: you're you are" will replace any contraction ‘you’re’, with ‘you are’, prior to any attempt to generate a response.

6.3 Keywords

Next, the individual words are compared to all keywords listed in the script file. The syntax of a keyword listing is:

"key: < keyword > < numerical weight >"

Example: "key: forget 4;

When the parser encounters the first match, it stores the keyword and its weight. If it later matches another keyword with a higher numerical weight, it will replace the stored keyword with the newer, higher ranking word. Thus the algorithm always matches the first word of the statement where the numerical weight of the matched keyword is greater than or equal to any other keyword matched within the statement. When no matches are found, a special keyword, "xnone" is automatically matched, providing a path to reconstruct a reply to the user.
6.4 Decomposition Rules

Every keyword entry in the script file is followed by at least one [decomp:] (decomposition) statement, and each [decomp:] is, in turn, followed by at least one [reasm:] (reassemble rule). Look at the following example.

key: my 2
    decomp: $ * my *
          reasm: Lets discuss further why your (2)
          reasm: Earlier you said your (2)
          reasm: But your (2)
    decomp: * my* @family *
          reasm: Tell me more about your family.
          reasm: Who else in your family (4)
          reasm: Your (3)
    decomp: * my *
          reasm: Your (2)
          reasm: Why do you say your (2)
          reasm: Does that suggest anything else which belongs to you.

This example contains the heart of the process by which Eliza reconstructs a response from the parsed statement, and contains most of the script constructs. First, whenever the keyword search returns ‘my’ as the highest-ranking keyword in the statement, the parser will next look at the [decomp:] statements following the keyword script entry. These entries act as secondary keywords or rules to further parse the statement.

6.5 Memory Flag

The ($) in the first decomposition rule means that whatever follows the first * must be at the very beginning of the input statement. In this case, ‘my’ is the same as the primary keyword, so we are simply looking for any sentence that starts with ‘my’. Therefore this rule only matches if the keyword is at the head of the linked list containing the parsed statement elements.
Additionally, all such responses will be stored in memory before being used. If another keyword exists in the input sentence, the program will create a response using ‘my’ and store it for later use before looking for another appropriate response to immediately return to the user. This allows Eliza to store up appropriate responses against the possibility that a future input generates no response. It also allows Eliza to remember earlier parts of the conversation.

6.6 Input String Indices

Note the imbedded numbers in parentheses. These references allow a script to use all or part of a user’s original input sentence within the generated response, (See Diagram on next page). Where (1) appears in a response the program will substitute the entire sentence input by the user, changing only those words within the sentence effected by post transformations. Where (2) appears, the program will apply post transforms and use only the portion of the input following the sub-key used. Where (3) appears and when a sub-key contains an additional *, as in * my* @family *, the sentence portion used starts from and includes the family synonym list member and continues to the end of the sentence. Finally, where (4) appears, the program replaces it with whatever follows the last element of a compound sub-key. The symbol * can be thought of as meaning ‘0 or more characters’ in the position indicated.

The second decomposition rule requires that the primary keyword ‘my’ must be followed by some member of the synonym list named family. Synonym lists are explained in the next section. Any number of words may intervene between the primary keyword and the synonym list member, this is denoted by the * attached to the end of the keyword. If that * was omitted, then the keyword would have to be followed immediately by a list member.

Finally, the last decomposition rule adds no additional conditions to the initial keyword match. It is a kind of default, when neither of the preceding rules is matched. If a given keyword has no additional search conditions attached, it is still necessary to include one decomposition rule to inform the
Figure 1: Reassembly of the indices
program that nothing else is needed. The default rule is "decomp: *
and is followed directly by all reassembly instructions for that keyword.

Decomposition rules are considered in order of listing. If the first rule
is not matched, control moves to the second. When a decomposition rule
is matched, the reassembly rules, immediately following, will reconstruct a
reply based on the linked list containing the parsed statement entered by the
user. On initialization of the program, each of the decomposition rules was
assigned a linked-list, (vector), in which to store the associated reassembly
rules which immediately follow it in the script. Each time a decomposition
rule is matched, the next available reassembly rule is used. After a reassembly
rule is used, control indexes to the next rule in the list. When all reassembly
rules have been used, the cycle just starts over. This will not necessarily
produce the same answer, because reassembly rules can use portions of the
users input statement in the construction of a reply. The numbers imbedded
in the reassembly rules indicate portions of the input statement that are to
be added in to the hard coded parts of the reassembly rules to provide the
final reply to the user.

6.7 Synonym Lists

The synonym lists are part of Eliza’s script file, and begin with the word
"synon". The action of a Synonym List is to convert any input word matching
a list element is transformed to the first word in the list. The first word in
any Synonym List should normally be a keyword or sub-key, (Decomposition
Rule), thus allowing the program to reply about some group concept rather
than just individual words. Different versions of Eliza use synonym lists in
different ways. In several of the later iterations, if any word in the sentence
matches any word in a synonym list, the word is replaced by the first word
of the synonym list. For example, the list:

"synon: family mother mom father dad sister brother wife children child",
is included in the Eliza script. Thus, if the sentence, “My mother and father
love me and my brother” is entered into one of these later Eliza programs, it
will be rendered by the pre-parser as “My family and family love me and my family.” At first, this might seem strange, but if the programmer wants to respond to any use of a family member, it makes sense. Later in those Eliza scripts, "family" would be used as a keyword, and appropriate responses, based only on the mention of a family member, would be returned. In the original Eliza, no such substitution is used. Synonym lists are merely a way of extending the pattern matching ability of the parser. Eliza looks normally for a single word key, but may then look for sub-keys. For example, if the keyword "my" is found, a second level key, called a ‘decomp’ will provide additional words to match. One such decomp is @family, which looks for any word matching the synonym list starting with family. At the first mention of ‘my’ followed by a family member, Eliza will reply to the user with, “Tell me more about your family”. This is, in fact, an appropriate response, considering Eliza is based on a Rogarian psychoanalyst. In psychoanalysis, the analyst’s participation in a conversation is minimal, saying only enough to keep the patient talking. Thus, this model is easily made to appear intelligent. More modern versions of Eliza, written in languages more adapted to parsing, like Perl, can greatly extend the power of Eliza’s parsing routines. Other constructs used by Eliza include pre: and post: which allow the program to substitute one word for another. These are matched pairs.

6.8 Post-Transformations

The post: transformation is designed to switch words during after the keyword search, and while the reply is being built. This allows the response to change some sentence components from first person to second person. The need for this should be obvious. If a user enters the string, “I am always sleepy after I eat”, Eliza will notice the keyword always, and might construct an appropriate response by asking a question based on the input of some condition that always happens. The reconstruction would be to say something like “Why are you” + [always] + remainder of sentence from keyword ‘always’. Thus generating, “Why are you always sleepy after I eat”. You will
notice that the case, first person, of the remainder of the sentence is wrong. By employing a (post:) transformation to the remainder of the sentence, all first person pronouns are replaced with second person, and all second person pronouns are replaced with first person pronouns. This would result in the response, “Why are you always sleepy after you eat”, which is correctly phrased. Even though I can easily construct sentences that will result in incorrect grammar, this approach is more often correct than wrong.

6.9 Emotion Triggers

Now that Eliza’s parsing system is known, the Emotion layer can be added by simply comparing keywords parsed against lists of trigger words. Like the original Eliza, I will set these up as synonym lists. If any keyword is matched in the angry list, the emotion interface will prepend the number 1 to every keyword found. If no matches are found in the angry-list, the sad-list is checked. If no matches are found, the reflective-list is checked. For a sad-list match, a 2 is prepended to each keyword found, or a 3 if there is a reflective-list match.

The original Eliza script was comprised of only 450 lines and less than 40 keywords. The new script will be expanded to include four keywords for each one in the original script. For each original keyword, I will add the same word with the numbers 1, 2, and 3 prepended, and following each of these new keywords will be reconstructions for responses that match each given emotion. Not all keywords need to vary by emotion, so a script "goto" statement, sending control back to the original keyword, will follow some. To speed the construction of an expanded script, a process that could take enormous amounts of time, I have collected a number of freeware versions of Eliza, and will recombine these scripts in a manner that suits my implementation.

For a more detailed description of script elements and environment variables consult the included Eliza Script Editor Manual.
7 Eliza Implementation

7.1 Scripts

My Java version of Eliza is based on use of Script Objects. A Script is composed of Keyword Objects, PrePost Objects, Synonym list Objects, MoodTrigger Objects and assorted environment variables. Each keyword object has three sub-key objects, one for each possible mood state. In turn, each sub-key object contains a set of possible responses appropriate to the triple of key, sub-key and mood.

7.2 File System

The program’s ability to rewrite scripts and change environment variables on the fly is provided by a file modification system. Whenever the Script Editor is instantiated, it reads an environment file that contains the name of all existing script files plus the currently active script. The existing script files are loaded into a choice box that defaults to the current script. The
user must first select the script he wishes to use as a base before any script modifications are allowed. Once a script is locked in, a new Script Object is instantiated and loaded from the selected file. After the chosen script is modified, it can be saved and at the users discretion named as the active script. By using the Script Editor, scripts can be changes without restarting the program.

7.3 Emotion Layer

Emotion is implemented by providing a separate set of sub-keys and responses for each keyword mood combination. While this effectively triples the Script size, it also allows complete flexibility of response by mood. The translation from file to script object is based on tagging each file keyword with a mood symbol. The mood symbol is subsequently stripped off when it is loaded into the Script Object and reapplied when writing any Script back to a file. An initial mood is determined by random at startup. The probabilities of each startup mood are adjustable with the Script Editor.

Each Script contains a list of Mood Triggers that can cause the program to modify its mood variables and state. These lists can be manipulated with the Script Editor. Each user-input sentence is preprocessed to determine the presence of mood triggers. If multiple triggers are encountered, each is processed in order according to mood increment and decrements rules and timers that may again be set by the user. The diagrams following detail the actions of Mood Logic, Program Environment, and the input parser on user input.

7.4 Display

Possibly the most difficult part of the implementation, the linkage of the Eliza application to the Haptek ActiveX control is a newly emerging technology. Haptek Inc., the developer or the ActiveX control provides Java Script wrappers to port the control to Java Applets but was unsuccessful in earlier attempts at porting to Java Console Applications. Sandbox restrictions and
Figure 3: Program Data Flow
the need to write files eliminate Applets as a possible solution. Another area of investigation was the possibility of converting the Java classes to Java Beans that are easily ported to the Windows API with the JINI interface. Here again, the reading, writing and modification of files by the Script Editor class are not easily expressible as a bean.

I created an early Applet version of Eliza without the mood interface or script toolkit.

Some of my earliest trials involved use of the Microsoft ActiveX Template Library, (ATL). Java Peer Classes were used to provide an integer reference to a Java Canvas to the Windows API. This allowed an Active X Control to be loaded on a Canvas and embedded in a Window API Frame. While I was successful in combining certain standard ActiveX controls like MS Calendar and MS Internet Explorer into these applications, I was unable to get the Haptek control to load. I abandoned this area of investigation when I discovered that Sun Microsystems had deprecated the Peer classes involved,
Figure 5: Logic for Mood Input

Program Environment Workflow on Input

Figure 6: Program Flow
Figure 7: Parser Flow
Figure 8: Early version of the Eliza applet
fearing that later removal of the classes would leave the Eliza application unusable.

My final area of research involved generating Java wrappers for the Haptek ActiveX control with IBM’s Bridge2Java, an alpha release downloaded from AlphaWorks, IBM’s alpha release site. The generated wrappers allowed the ActiveX control to be loaded in a standard Java Console Application. This approach finally produced a usable interface for the application.

To better understand the functions of the wrapper classes generated by the IBM application, I de-compiled the Bridge2Java classes using a Mocha Java de-compiler. I was surprised to find that the IBM used the same deprecated Peer classes I used in my earlier investigation. Given reports of IBM’s plans to integrate this particular functionality into their Visual Age for Java Environment, I suspect that Sun is not planning changes that will completely
end the functionality of these classes. As a precaution, I will extract the Peer classes used by my Eliza implementation from the JDK1.3 jar file and include them with the application distribution.

The final version of the Eliza application will pass response strings to the speech engine of the Haptek control. Additionally, the application will set background lighting in the ActiveX display to provide a visual indicator of the current mood. Angry moods will have red background lighting while reflective will have green. Normal mood will involve natural white background light.

Numerous voice components and traits can modify the speech engine used. An angry state will be associated with a speech rate and pitch that are slightly faster than normal. Reflective mood states will be associated with voice traits slightly slower in rate and lower in pitch. The talking heads can also read expression files that program the moment to moment facial expressions of the morphs. Many of the details of this part of the Haptek API are published and will allow provision of three different facial script files to match the current mood state of the application.

### 7.5 Toolkit

Documentation of the Eliza Script Editor toolkit is included as a separate ‘User Manual’.

### 8 Conclusions and Recommendations

The project was successful in demonstrating that modular additions to Eliza’s basic program can extend the impression of Artificial Intelligence. I say the ‘impression’ because no such program can be considered intelligent until we reach the point of being able to parse a user’s input into linguistic concepts, rather than mere keywords. However, the addition mood states coupled with facial expressions and vocal changes which non-verbally supported those moods, substantially improved the perception of intelligence.
REFERENCES

To a large extent, creation of believable Eliza scripts is a matter of trial and error. The Script Editor developed in this project should greatly ease this task, allowing users to quickly modify and test new script elements.

Since the basic Eliza script used in this project was small, the mood extensions had little effect on the program performance. However, to fully implement all moods normally associated with human intelligence, as well as combinations and degrees of those moods would raise some interesting questions.

- What is the smallest subset of human moods from which a machine could generate most normal human mood states?

- How can a program handle moods of varying levels and combinations without requiring an enormous number of discrete response sets covering every possible combination?

An interesting next step in the integration of emotions into an Eliza like program might involve use of the newly emerging Cyc Project. Once public access to the Cyc database is available, it would provide an excellent opportunity to replace Eliza’s current keyword parser with a concept parser based on the rules system and language databases at Cyc.

One open question and possible area of future investigation involves whether it is possible to replace discrete mood response sets with one that performs fuzzy logic selection of sentence modifiers based on a set of mood level inputs. That is, the program would generate only a single response and then modify that response based on an input array of mood levels. These mood levels would be dynamic, changing in relation to user input and program parameters.

References

REFERENCES


