CHAPTER 5

# ELIZA: Dialog with a Machine

*It is said that to explain is to explain away.*

—Joseph Weizenbaum
MIT computer scientist

This chapter and the rest of part I will examine three more well-known AI programs of the 1960s. ELIZA held a conversation with the user in which it simulated a psychotherapist. STUDENT solved word problems of the kind found in high school algebra books, and MACSYMA solved a variety of symbolic mathematical problems, including differential and integral calculus. We will develop versions of the first two programs that duplicate most of the essential features, but for the third we will implement only a tiny fraction of the original program's capabilities.

All three programs make heavy use of a technique called pattern matching. Part I serves to show the versatility—and also the limitations—of this technique.

Of the three programs, the first two process input in plain English, and the last two solve nontrivial problems in mathematics, so there is some basis for describing them as being "intelligent." On the other hand, we shall see that this intelligence is largely an illusion, and that ELIZA in particular was actually designed to demonstrate this illusion, not to be a "serious" AI program.

ELIZA was one of the first programs to feature English output as well as input. The program was named after the heroine of *Pygmalion,* who was taught to speak proper English by a dedicated teacher. ELIZA's principal developer, MIT professor Joseph Weizenbaum, published a paper on ELIZA in the January 1966 issue of the *Communications of the Association for Computing Machinery*. The introduction to that paper is reproduced in its entirety here:

> *It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming, especially in what is called heuristic programming and artificial intelligence. For in those realms machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer. But once a particular program is unmasked, once its inner workings are explained in language sufficiently plain to induce understanding, its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible. The observer says to himself, "I could have written that." With that thought he moves the program in question from the shelf marked "intelligent," to that reserved for curios, fit to be discussed only with people less enlightened than he.*
>
> *The object of this paper is to cause just such a re-evaluation of the program about to be "explained." Few programs ever needed it more.*

Let me point out in passing that the main object of this book is to cause the reader to say to him or herself, "I could have written that." It is hoped that in some cases this will be because the reader has gained something, and not solely because the programs in question have lost their ability to dazzle.

Let's start our investigation of ELIZA by looking at a sample dialog from the Weizenbaum article. ELIZA was designed to be instructed by a variety of scripts, or patterns of interaction. In fact it had several scripts, including ones in German and Welsh. But the only script that was used to any extent was in English, and it made ELIZA emulate a Rogerian psychoanalyst. Rogerians are "nondirective"—they try to let the patient reveal himself. This is an easy script for ELIZA to follow, because it involves being passive and reacting, rather than volunteering new information. In the listing that follows, the user's input is in mixed case, and ELIZA'S response is in capitals:

```
Men are all alike.
IN WHAT WAY
They're always bugging us about something or another.
CAN YOU BE MORE SPECIFIC
Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
```

```
It's true.  I am unhappy.
DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
I need some help, that much seems certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps I could learn to get along with my mother.
TELL ME MORE ABOUT YOUR FAMILY
```

This seems quite impressive; the program is responding reasonably to a variety of inputs and appears to be exhibiting true "understanding." But, as Weizenbaum explains, this appearance is an illusion, and the program has finessed the difficult task of understanding by carefully recognizing, transforming, and echoing pieces of the input. The procedure is to look for specific patterns, based on a key word or words in the input. For example, whenever the program sees the words "alike" or "same," it prints a message asking about similarity, in this case, "IN WHAT **WAY.**" Other patterns are more complicated. In the next to last exchange, ELIZA first throws away everything after the comma, and then uses a pattern that looks for input of the form "I need X and transforms it to the output "What would it mean to you if you got X," where X is the phrase "some help" in this case. The great advantage of this approach is that the response will make sense for a variety of inputs: "I need a vacation" will elicit "What would it mean to you if you got a vacation." This will work automatically, without having to tell the program the difference between "some help" and "a vacation." The disadvantage is that the program will sometimes make silly responses: "I need this like a hole in the head will be answered with "What would it mean to you if you got this like a hole in the head." In fact, the program only makes any sense at all if it is receiving input from someone who makes sense—and someone who interprets its answers charitably. When given "I need gibble plix zzq," ELIZA would reply "What would it mean to you if you got gibble plix zzq," without any inkling that something is wrong. Another example of the original ELIZA program has it communicating with another program, PARRY, which was written by Dr. Kenneth Colby to mimic paranoid behavior. Part of the conversation follows, with ELIZA again in capitals and PARRY in mixed case:

```
I don't understand your motives.
WHY DON'T YOU UNDERSTAND MY MOTIVES?
Shouldn't I be?
WHY DO YOU SAY THAT JUST NOW?
That's just the way it is.
SUPPOSE THAT'S JUST THE WAY IT WERE NOT.
So?
I AM NOT SURE I UNDERSTAND YOU FULLY.
You harp on the same questions too much.
WHAT DOES THAT SIMILARITY SUGGEST TO YOU?
I don't get you.
DO YOU WISH THAT YOU DID GET I?
```

This suggests that neither ELIZA nor PARRY had a deep understanding of the dialog. When there is a human to take on most of the burden of carrying forth the conversation, these programs can be convincing, but when the human is taken out of the conversation, the weaknesses of the programs are revealed.

## 5.1 Describing and Specifying ELIZA

Now that we have an idea of what ELIZA is like, we can begin the description and specification of the program, and eventually move to the implementation and debugging.

The ELIZA algorithm can be described simply as: (1) read an input, (2) find a pattern that matches the input, (3) transform the input into a response, and (4) print the response. These four steps are repeated for each input.

The specification and implementation of steps (1) and (4) are trivial: for (1), use the built-in read function to read a list of words, and for (4) use `print` to print the list of words in the response.

Of course, there are some drawbacks to this specification. The user will have to type a real list—using parentheses—and the user can't use characters that are special to read, like quotation marks, commas, and periods. So our input won't be as unconstrained as in the sample dialog, but that's a small price to pay for the convenience of having half of the problem neatly solved.

## 5.2 Pattern Matching

The hard part comes with steps (2) and (3)—this notion of pattern matching and transformation. There are four things to be concerned with: a general pattern and response, and a specific input and transformation of that input. Since we have agreed to represent the input as a list, it makes sense for the other components to be lists too. For example, we might have:

```
Pattern: (i need a X)
Response: (what would it mean to you if you got a X ?)

Input: (i need a vacation)
Transformation: (what would it mean to you if you got a vacation ?)
```

The pattern matcher must match the literals `i` with `i`, `need` with `need`, and `a` with `a`, as well as match the variable `X` with `vacation`. This presupposes that there is some way of deciding that `X` is a variable and that `need` is not. We must then arrange to substitute `vacation` for `X` within the response, in order to get the final transformation.

Ignoring for a moment the problem of transforming the pattern into the response, we can see that this notion of pattern matching is just a generalization of the Lisp function equal. Below we show the function simple-equal, which is like the built-in function equal,[1] and the function pat-match, which is extended to handle pattern-matching variables:

```
(defun simple-equal (x y)
  "Are x and y equal?  (Don't check inside strings.)"
  (if (or (atom x) (atom y))
      (eql x y)
      (and (simple-equal (first x) (first y))
           (simple-equal (rest x) (rest y)))))

(defun pat-match (pattern input)
  "Does pattern match input?  Any variable can match anything."
  (if (variable-p pattern)
      t
      (if (or (atom pattern) (atom input))
          (eql pattern input)
          (and (pat-match (first pattern) (first input))
               (pat-match (rest pattern) (rest input))))))
```

**Exercise 5.1 [s]** Would it be a good idea to replace the complex and form in pat-match with the simpler (every #'pat-match pattern input)? it's could

Before we can go on, we need to decide on an implementation for pattern-matching variables. We could, for instance, say that only a certain set of symbols, such as {X,Y,Z}, are variables. Alternately, we could define a structure of type variable, but then we'd have to type something verbose like (make-variable :name 'X) every time we wanted one. Another choice would be to use symbols, but to distinguish variables from constants by the name of the symbol. For example, in Prolog, variables start with capital letters and constants with lowercase. But Common Lisp is case-insensitive, so that won't work. Instead, there is a tradition in Lisp-based AI programs to have variables be symbols that start with the question mark character.

So far we have dealt with symbols as atoms—objects with no internal structure. But things are always more complicated than they first appear and, as in Lisp as in physics, it turns out that even atoms have components. In particular, symbols have names, which are strings and are accessible through the symbol-name function. Strings in turn have elements that are characters, accessible through the function char. The character '?' is denoted by the self-evaluating escape sequence #\?. So the predicate variable-p can be defined as follows, and we now have a complete pattern matcher:

[1] The difference is that simple-equal does not handle strings.

```
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x) (equal (char (symbol-name x) 0) #\?)))

> (pat-match '(I need a ?X) '(I need a vacation))
T

> (pat-match '(I need a ?X) '(I really need a vacation))
NIL
```

In each case we get the right answer, but we don't get any indication of what ?X is, so we couldn't substitute it into the response. We need to modify pat-match to return some kind of table of variables and corresponding values. In making this choice, the experienced Common Lisp programmer can save some time by being opportunistic: recognizing when there is an existing function that will do a large part of the task at hand. What we want is to substitute values for variables throughout the response. The alert programmer could refer to the index of this book or the Common Lisp reference manual and find the functions substitute, subst, and sublis. All of these substitute some new expression for an old one within an expression. It turns out that sublis is most appropriate because it is the only one that allows us to make several substitutions all at once. sublis takes two arguments, the first a list of old-new pairs, and the second an expression in which to make the substitutions. For each one of the pairs, the car is replaced by the cdr. In other words, we would form each pair with something like (cons old new). (Such a list of pairs is known as an association list, or a-list, because it associates keys with values. See section 3.6.) In terms of the example above, we would use:

```
> (sublis '((?X . vacation))
          '(what would it mean to you if you got a ?X ?))
(WHAT WOULD IT MEAN TO YOU IF YOU GOT A VACATION ?)
```

Now we need to arrange for pat-match to return an a-list, rather than just T for success. Here's a first attempt:

```
(defun pat-match (pattern input)
  "Does pattern match input? WARNING: buggy version."
  (if (variable-p pattern)
      (list (cons pattern input))
      (if (or (atom pattern) (atom input))
          (eql pattern input)
          (append (pat-match (first pattern) (first input))
                  (pat-match (rest pattern) (rest input))))))
```

This implementation looks reasonable: it returns an a-list of one element if the pattern is a variable, and it appends alists if the pattern and input are both lists. However,

there are several problems. First, the test (eql pattern input) may return T, which is not a list, so append will complain. Second, the same test might return nil, which should indicate failure, but it will just be treated as a list, and will be appended to the rest of the answer. Third, we haven't distinguished between the case where the match fails—and returns nil—versus the case where everything matches, but there are no variables, so it returns the null a-list. (This is the semipredicate problem discussed on page 127.) Fourth, we want the bindings of variables to agree—if ?X is used twice in the pattern, we don't want it to match two different values in the input. Finally, it is inefficient for pat-match to check both the first and rest of lists, even when the corresponding first parts fail to match. (Isn't it amazing that there could be five bugs in a seven-line function?)

We can resolve these problems by agreeing on two major conventions. First, it is very convenient to make pat-match a true predicate, so we will agree that it returns nil only to indicate failure. That means that we will need a non-nil value to represent the empty binding list. Second, if we are going to be consistent about the values of variables, then the first will have to know what the rest is doing. We can accomplish this by passing the binding list as a third argument to pat-match. We make it an optional argument, because we want to be able to say simply (pat-match a b).

To abstract away from these implementation decisions, we define the constants fail and no-bindings to represent the two problematic return values. The special form defconstant is used to indicate that these values will not change. (It is customary to give special variables names beginning and ending with asterisks, but this convention usually is not followed for constants. The reasoning is that asterisks shout out, "Careful! I may be changed by something outside of this lexical scope." Constants, of course, will not be changed.)

```
(defconstant fail nil "Indicates pat-match failure")

(defconstant no-bindings '((t . t))
  "Indicates pat-match success, with no variables.")
```

Next, we abstract away from assoc by introducing the following four functions:

```
(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))
```

```
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val) bindings))
```

Now that variables and bindings are defined, pat-match is easy. It consists of five cases. First, if the binding list is fail, then the match fails (because some previous match must have failed). If the pattern is a single variable, then the match returns whatever match-variable returns; either the existing binding list, an extended one, or fail. Next, if both pattern and input are lists, we first call pat-match recursively on the first element of each list. This returns a binding list (or fail), which we use to match the rest of the lists. This is the only case that invokes a nontrivial function, so it is a good idea to informally prove that the function will terminate: each of the two recursive calls reduces the size of both pattern and input, and pat-match checks the case of atomic patterns and inputs, so the function as a whole must eventually return an answer (unless both pattern and input are of infinite size). If none of these four cases succeeds, then the match fails.

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((and (consp pattern) (consp input))
         (pat-match (rest pattern) (rest input)
                    (pat-match (first pattern) (first input)
                               bindings)))
        (t fail)))

(defun match-variable (var input bindings)
  "Does VAR match input?  Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
          ((equal input (binding-val binding)) bindings)
          (t fail))))
```

We can now test pat-match and see how it works:

```
> (pat-match '(i need a ?X) '(i need a vacation))
((?X . VACATION) (T . T))
```

The answer is a list of variable bindings in dotted pair notation; each element of the list is a (variable . value) pair. The (T . T) is a remnant from no-bindings. It does no real harm, but we can eliminate it by making extend-bindings a little more complicated:

```
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy no-bindings
        (if (eq bindings no-bindings)
            nil
            bindings)))
```

```
> (sublis (pat-match '(i need a ?X) '(i need a vacation))
          '(what would it mean to you if you got a ?X ?))
(WHAT WOULD IT MEAN TO YOU IF YOU GOT A VACATION ?)

> (pat-match '(i need a ?X) '(i really need a vacation))
NIL

> (pat-match '(this is easy) '(this is easy))
((T . T))

> (pat-match '(?X is ?X) '((2 + 2) is 4))
NIL

> (pat-match '(?X is ?X) '((2 + 2) is (2 + 2)))
((?X 2 + 2))

> (pat-match '(?P need . ?X) '(i need a long vacation))
((?X A LONG VACATION) (?P . I))
```

Notice the distinction between `NIL` and `((T . T))`. The latter means that the match succeeded, but there were no bindings to return. Also, remember that `(?X 2 + 2)` means the same as `(?X . (2 + 2))`.

A more powerful implementation of `pat-match` is given in chapter 6. Yet another implementation is given in section 10.4. It is more efficient but more cumbersome to use.

## 5.3 Segment Pattern Matching

In the pattern `(?P need . ?X)`, the variable `?X` matches the rest of the input list, regardless of its length. This is in contrast to `?P`, which can only match a single element, namely, the first element of the input. For many applications of pattern matching, this is fine; we only want to match corresponding elements. However, ELIZA is somewhat different in that we need to account for variables in any position that match a sequence of items in the input. We will call such variables segment variables. We will need a notation to differentiate segment variables from normal

variables. The possibilities fall into two classes: either we use atoms to represent segment variables and distinguish them by some spelling convention (as we did to distinguish variables from constants) or we use a nonatomic construct. We will choose the latter, using a list of the form (?* variable) to denote segment variables. The symbol ?* is chosen because it combines the notion of variable with the Kleene-star notation. So, the behavior we want from pat-match is now:

```
> (pat-match '((?* ?p) need (?* ?x))
             '(Mr Hulot and I need a vacation))
((?P MR HULOT AND I) (?X A VACATION))
```

In other words, when both pattern and input are lists and the first element of the pattern is a segment variable, then the variable will match some initial part of the input, and the rest of the pattern will attempt to match the rest. We can update pat-match to account for this by adding a single cond-clause. Defining the predicate to test for segment variables is also easy:

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((segment-pattern-p pattern)                ; ***
         (segment-match pattern input bindings))    ; ***
        ((and (consp pattern) (consp input))
         (pat-match (rest pattern) (rest input)
                    (pat-match (first pattern) (first input)
                               bindings)))
        (t fail)))

(defun segment-pattern-p (pattern)
  "Is this a segment matching pattern: ((?* var) . pat)"
  (and (consp pattern)
       (starts-with (first pattern) '?*)))
```

In writing segment-match, the important question is how much of the input the segment variable should match. One answer is to look at the next element of the pattern (the one after the segment variable) and see at what position it occurs in the input. If it doesn't occur, the total pattern can never match, and we should fail. If it does occur, call its position pos. We will want to match the variable against the initial part of the input, up to pos. But first we have to see if the rest of the pattern matches the rest of the input. This is done by a recursive call to pat-match. Let the result of this recursive call be named **b2**. If **b2** succeeds, then we go ahead and match the segment variable against the initial subsequence.

The tricky part is when b2 fails. We don't want to give up completely, because it may be that if the segment variable matched a longer subsequence of the input, then the rest of the pattern would match the rest of the input. So what we want is to try segment-match again, but forcing it to consider a longer match for the variable. This is done by introducing an optional parameter, start, which is initially 0 and is increased with each failure. Notice that this policy rules out the possibility of any kind of variable following a segment variable. (Later we will remove this constraint.)

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        ;; We assume that pat starts with a constant
        ;; In other words, a pattern can't have 2 consecutive vars
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
              fail
              (let ((b2 (pat-match pat (subseq input pos) bindings)))
                ;; If this match failed, try another longer one
                ;; If it worked, check that the variables match
                (if (eq b2 fail)
                    (segment-match pattern input bindings (+ pos 1))
                    (match-variable var (subseq input 0 pos) b2)))))))))
```

Some examples of segment matching follow:

```
> (pat-match '((?* ?p) need (?* ?x))
             '(Mr Hulot and I need a vacation))
((?P MR HULOT AND I) (?X A VACATION))

> (pat-match '((?* ?x) is a (?* ?y)) '(what he is is a fool))
((?X WHAT HE IS) (?Y FOOL))
```

The first of these examples shows a fairly simple case: ?p matches everything up to need, and ?x matches the rest. The next example involves the more complicated backup case. First ?x matches everything up to the first is (this is position 2, since counting starts at 0 in Common Lisp). But then the pattern a fails to match the input is, so segment-match tries again with starting position **3.** This time everything works; is matches is, a matches a, and (?* ?y) matches fool.

Unfortunately, this version of segment-match does not match as much as it should. Consider the following example:

```
> (pat-match '((?* ?x) a b (?* ?x)) '(1 2 a b a b 1 2 a b)) ⇒ NIL
```

This fails because ?x is matched against the subsequence (1 2), and then the remaining pattern succesfully matches the remaining input, but the final call to match-variable fails, because ?x has two different values. The fix is to call match-variable before testing whether the b2 fails, so that we will be sure to try segment-match again with a longer match no matter what the cause of the failure.

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        ;; We assume that pat starts with a constant
        ;; In other words, a pattern can't have 2 consecutive vars
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
              fail
              (let ((b2 (pat-match
                         pat (subseq input pos)
                         (match-variable var (subseq input 0 pos)
                                         bindings))))
                ;; If this match failed, try another longer one
                (if (eq b2 fail)
                    (segment-match pattern input bindings (+ pos 1))
                    b2)))))))
```

Now we see that the match goes through:

```
> (pat-match '((?* ?x) a b (?*?x)) '(1 2 a b a b 1 2 a b))
((?X 1 2 A B))
```

Note that this version of segment-match tries the shortest possible match first. It would also be possible to try the longest match first.

## 5.4 The ELIZA Program: A Rule-Based Translator

Now that we have a working pattern matcher, we need some patterns to match. What's more, we want the patterns to be associated with responses. We can do this by inventing a data structure called a rule, which consists of a pattern and one or more associated responses. These are rules in the sense that they assert, "If you see A, then respond with B or C, chosen at random." We will choose the simplest possible implementation for rules: as lists, where the first element is the pattern and the rest is a list of responses:

```
(defun rule-pattern (rule) (first rule))
(defun rule-responses (rule) (rest rule))
```

Here's an example of a rule:

```
(((?* ?x) I want (?* ?y))
 (What would it mean if you got ?y)
 (Why do you want ?y)
 (Suppose you got ?y soon))
```

When applied to the input (I want to test this program), this rule (when interpreted by the ELIZA program) would pick a response at random, substitute in the value of ?y, and respond with, say, (why do you want to test this program).

Now that we know what an individual rule will do, we need to decide how to handle a set of rules. If ELIZA is to be of any interest, it will have to have a variety of responses. So several rules may all be applicable to the same input. One possibility would be to choose a rule at random from among the rules having patterns that match the input.

Another possibility is just to accept the first rule that matches. This implies that the rules form an ordered list, rather than an unordered set. The clever ELIZA rule writer can take advantage of this ordering and arrange for the most specific rules to come first, while more vague rules are near the end of the list.

The original ELIZA had a system where each rule had a priority number associated with it. The matching rule with the highest priority was chosen. Note that putting the rules in order achieves the same effect as having a priority number on each rule: the first rule implicitly has the highest priority, the second rule is next highest, and so on.

Here is a short list of rules, selected from Weizenbaum's original article, but with the form of the rules updated to the form we are using. The answer to exercise 5.19 contains a longer list of rules.

```
(defparameter *eliza-rules*
 '((((?* ?x) hello (?* ?y))
    (How do you do.  Please state your problem.))
   (((?* ?x) I want (?* ?y))
    (What would it mean if you got ?y)
    (Why do you want ?y) (Suppose you got ?y soon))
   (((?* ?x) if (?* ?y))
    (Do you really think its likely that ?y) (Do you wish that ?y)
    (What do you think about ?y) (Really-- if ?y))
   (((?* ?x) no (?* ?y))
    (Why not?) (You are being a bit negative)
    (Are you saying "NO" just to be negative?))
   (((?* ?x) I was (?* ?y))
    (Were you really?) (Perhaps I already knew you were ?y)
    (Why do you tell me you were ?y now?))
   (((?* ?x) I feel (?* ?y))
    (Do you often feel ?y ?))
   (((?* ?x) I felt (?* ?y))
    (What other feelings do you have?))))
```

Finally we are ready to define ELIZA proper. As we said earlier, the main program should be a loop that reads input, transforms it, and prints the result. Transformation is done primarily by finding some rule such that its pattern matches the input, and then substituting the variables into the rule's response. The program is summarized in figure 5.1.

There are a few minor complications. We print a prompt to tell the user to input something. We use the function flatten to insure that the output won't have imbedded lists after variable substitution. An important trick is to alter the input by swapping "you" for "me" and so on, since these terms are relative to the speaker. Here is the complete program:

```
(defun eliza ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)
    (write (flatten (use-eliza-rules (read))) :pretty t)))

(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (some #'(lambda (rule)
            (let ((result (pat-match (rule-pattern rule) input)))
              (if (not (eq result fail))
                  (sublis (switch-viewpoint result)
                          (random-elt (rule-responses rule))))))
        *eliza-rules*))
```

| | |
|---|---|
| | **Top-Level Function** |
| eliza | Respond to user input using pattern matching rules. |
| | **Special Variables** |
| *eliza-rules* | A list of transformation rules. |
| | **Data Types** |
| rule | An association of a pattern with a list of responses. |
| | **Functions** |
| eliza | Respond to user input using pattern matching rules. |
| use-eliza-rules | Find some rule with which to transform the input. |
| switch-viewpoint | Change I to you and vice versa, and so on. |
| flatten | Append together elements of a list. |
| | **Selected Common Lisp Functions** |
| sublis | Substitute elements into a tree. |
| | **Previously Defined Functions** |
| random-elt | Pick a random element from a list. (p.36) |
| pat-match | Match a pattern against an input. (p.160) |
| mappend | Append together the results of a mapcar. |

Figure 5.1: Glossary for the ELIZA Program

```
(defun switch-viewpoint (words)
  "Change I to you and vice versa, and so on."
  (sublis '((I . you) (you . I) (me . you) (am . are))
          words))
```

Note the use of write with the :pretty keyword true. This will give better formatted output in some cases. The program makes use of the previously defined random-elt, and flatten, which is defined here using mappend and mklist, a function that is defined in the InterLisp dialect but not in Common Lisp.

```
(defun flatten (the-list)
  "Append together elements (or lists) in the list."
  (mappend #'mklist the-list))

(defun mklist (x)
  "Return x if it is a list, otherwise (x)."
  (if (listp x)
      x
      (list x)))

(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))
```

```
(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

The actual ELIZA program had a few complications that don't show up in this version. First, there was an alias method for associating several words with the same pattern; both "mother" and "father" could be associated with the "family" pattern. There was also a synonym mechanism that would treat "don't" and "do not" or "everybody" and "everyone" the same. In the case of input with several comma-separated phrases, each phrase was processed separately, and the response with the highest priority was selected. In addition, the original ELIZA had a "memory" mechanism. When no pattern matched the input, it said something like "Tell me more about X," where X is some input given earlier. Finally, our version has fewer rules. Despite the shortcomings of our version, it can occasionally hold up its end of a conversation, as shown below:

```
> (eliza)
ELIZA> (hello there)
(HOW DO YOU DO. PLEASE STATE YOUR PROBLEM.)
ELIZA> (i want to test this program)
(WHAT WOULD IT MEAN IF YOU GOT TO TEST THIS PROGRAM)
ELIZA> (i could see if it works)
(DO YOU REALLY THINK ITS LIKELY THAT IT WORKS)
ELIZA> (no not really)
(ARE YOU SAYING "NO" JUST TO BE NEGATIVE?)
ELIZA> (no)
(ARE YOU SAYING "NO" JUST TO BE NEGATIVE?)
ELIZA> (forget it-- i was wondering how general the program is)
(WHY DO YOU TELL ME YOU WERE WONDERING HOW GENERAL THE PROGRAM IS NOW?)
ELIZA> (i felt like it)
(WHAT OTHER FEELINGS DO YOU HAVE?)
ELIZA> (i feel this is enough)
(DO YOU OFTEN FEEL THIS IS ENOUGH ?)
ELIZA> [Abort]
>
```

In the end, it is the technique that is important—not the program. ELIZA has been "explained away" and should rightfully be moved to the curio shelf. Pattern matching in general remains important technique, and we will see it again in subsequent chapters. The notion of a rule-based translator is also important. The problem of understanding English (and other languages) remains an important part of AI. Clearly, the problem of understanding English is not solved by ELIZA. In part V, we will address the problem again, using more sophisticated techniques.

## 5.5 History and References

As mentioned above, the original article describing ELIZA is Weizenbaum1966. Another dialog system using similar pattern-matching techniques is Kenneth Colby's (1975) PARRY. This program simulated the conversation of a paranoid person well enough to fool several professional psychologists. Although the pattern matching techniques were simple, the model of belief maintained by the system was much more sophisticated than ELIZA. Colby has suggested that dialog programs like ELIZA, augmented with some sort of belief model like PARRY, could be useful tools in treating mentally disturbed people. According to Colby, it would be inexpensive and effective to have patients converse with a specially designed program, one that could handle simple cases and alert doctors to patients that needed more help. Weizenbaum's book *Computer Power and Human Reason* (1976) discusses ELIZA and PARRY and takes a very critical view toward Colby's suggestion. Other interesting early work on dialog systems that model belief is reported by Allan Collins (1978) and Jamie Carbonell (1981).

## 5.6 Exercises

**Exercise 5.2 [m]** Experiment with this version of ELIZA. Show some exchanges where it performs well, and some where it fails. Try to characterize the difference. Which failures could be fixed by changing the rule set, which by changing the `pat-match` function (and the pattern language it defines), and which require a change to the `eliza` program itself?

**Exercise 5.3 [h]** Define a new set of rules that make ELIZA give stereotypical responses to some situation other than the doctor–patient relationship. Or, write a set of rules in a language other than English. Test and debug your new rule set.

**Exercise 5.4 [s]** We mentioned that our version of ELIZA cannot handle commas or double quote marks in the input. However, it seems to handle the apostrophe in both input and patterns. Explain.

**Exercise 5.5 [h]** Alter the input mechanism to handle commas and other punctuation characters. Also arrange so that the user doesn't have to type parentheses around the whole input expression. (Hint: this can only be done using some Lisp functions we have not seen yet. Look at `read-line` and `read-from-string`.)