9 Example: Doctor

Program file for this chapter: `doctor`

```
? doctor
Hello, I am the doctor.  What can i do for you?
Please end your remarks with an empty line.

My spirits are low.

Your spirits are low?

I'm really depressed.

I'm sorry to hear you are depressed.

It's because nobody likes me.

Really, nobody?

That's right.

I am not sure I understand you fully.

I'm lonely.

Is it because you are lonely that you came to me?

Of course.

Earlier you said your spirits are low.
```

This is the beginning of a dialogue with a program called `doctor`. The program simulates the behavior of a psychotherapist. People who interact with this program are

often fooled into thinking that the program understands what they are saying, although in fact the program does not understand at all. Instead, the program recognizes certain key words in a sentence, words like "problem" or "nobody." For each of these words, there is a list of *rules;* a rule consists of a pattern (of the sort used by the pattern matcher of Chapter 7) and a set of possible responses. For example, associated with the word "nobody" is the pattern

```
[# !a:in [everyone everybody nobody noone] #]
```

Corresponding to that pattern is a list of responses:

```
[[Really, :a?]
 [Surely not :a.]
 [Can you think of anyone in particular?]
 [Who, for example?]
 [You are thinking of a very special person.]
 [Who, may I ask?]
 [Someone special perhaps.]
 [You have a particular person in mind, don't you?]
 [Who do you think you're talking about?]
 [I suspect you're exaggerating a little.]]
```

The program has so many possible responses so that each time you use the word "nobody" (or "everybody," etc.) you get a different answer. Even though many of the answers have the same meaning, the variety in the wording helps to convince you that it's a real person at the other end of the conversation.

Many versions of this program have been written, but the first was written in 1966 by Joseph Weizenbaum, a professor of computer science at MIT. He called his program Eliza after Eliza Doolittle, a character in the play *Pygmalion* by George Bernard Shaw. Eliza started life poor and uneducated, with rough speech, but in the play she is taught to speak better. The program, too, can be taught rules to help it speak better.

> Because conversations must be about something, that is, because they must take place within some context, the program was constructed in a two-tier arrangement, the first tier consisting of the language analyzer and the second of a script. The script is a set of rules rather like those that might be given to an actor who is to use them to improvise around a certain theme. Thus Eliza could be given a script to enable it to maintain a conversation about cooking eggs or about managing a bank checking account, and so on. Each specific script thus enabled Eliza to play a specific conversational role.

> For my first experiment, I gave Eliza a script designed to permit it to play (I should really say parody) the role of a Rogerian psychotherapist engaged in an initial interview with a patient. The Rogerian psychotherapist is relatively easy to imitate because much of his technique consists of drawing his patient out by reflecting the patient's statements back to him...
>
> [Joseph Weizenbaum, *Computer Power and Human Reason* (Freeman, 1976), page 3.]

It has long been a popular Logo programming project to implement a small subset of `doctor`'s conversational ability through a program that embodies directly a few of the rules in Weizenbaum's Doctor script for Eliza. (See, for example, Harold Abelson, *Apple Logo* (BYTE Publications, 1982), page 158.) Home computers now have enough memory to permit the implementation in Logo of Weizenbaum's original two-tier approach. The program presented here is not a line-for-line translation of Eliza into Logo, but does embody the same fundamental strategy (namely pattern matching) as the original. The script is a close adaptation of Weizenbaum's version, via a Lisp version by Jon L. White.

You should try conversing with `doctor` yourself a few times. Whenever you get a response that seems linguistically bizarre, make a note of it. Later, as I'm talking about the way the program works, you can ask yourself how you would modify the script to eliminate the bad responses you've noted.

Eliza and Artificial Intelligence

When Eliza was first unveiled, many people considered it a major advance in the pursuit of *artificial intelligence:* the search for ways to make computers as intelligent as people. Especially to a person unfamiliar with computer programming, a conversation with the Doctor can seem very real indeed. But as Weizenbaum himself points out, the underlying techniques used in the program do not involve any real understanding of the conversation. The program "cheats."

Today there are language-understanding programs that use techniques much more sophisticated than the simple pattern matching of Eliza. The authors of some such programs maintain that they really do understand what they are saying and hearing, in a sense in which Eliza does not. Other people, including Weizenbaum, suggest that even these state-of-the-art artificial intelligence programs are merely cheating in more complex ways. What would it mean for a program "really" to understand a conversation? This is a deep question that would take too long to explore here. (I'll come back to it in the discussion of artificial intelligence in the third volume of this series.) If you're

interested, you should begin by reading Weizenbaum's book, *Computer Power and Human Reason*, from which I quoted a passage earlier. He argues not only that computers *cannot* do certain things, but also that people *should not* use computers for certain purposes even if it were possible. In particular, he is horrified at the suggestion, made even by some psychiatrists, that programs like Eliza should be used to provide therapy to actual patients.

Eliza's Linguistic Strategy

The program allows the user to enter one or more sentences, typing several lines if necessary. The first step in processing the user's remarks is to string the several lines into one long list. This list is a "sentence" in the Logo technical sense; that is, it's a list of words with no sublist structure. It may, however, contain one or more English sentences. The generation of a response involves several steps:

1. Find the punctuation in the input and use it to extract a single sentence to answer.
2. Find keywords in the sentence for which the script includes rules.
3. Apply word-for-word translations as specified by the script; these are primarily to convert first person ("I") to second person ("you") and vice versa.
4. Pick the highest priority keyword.
5. Find a rule for that keyword whose pattern matches the input sentence.
6. Choose a response according to that rule.

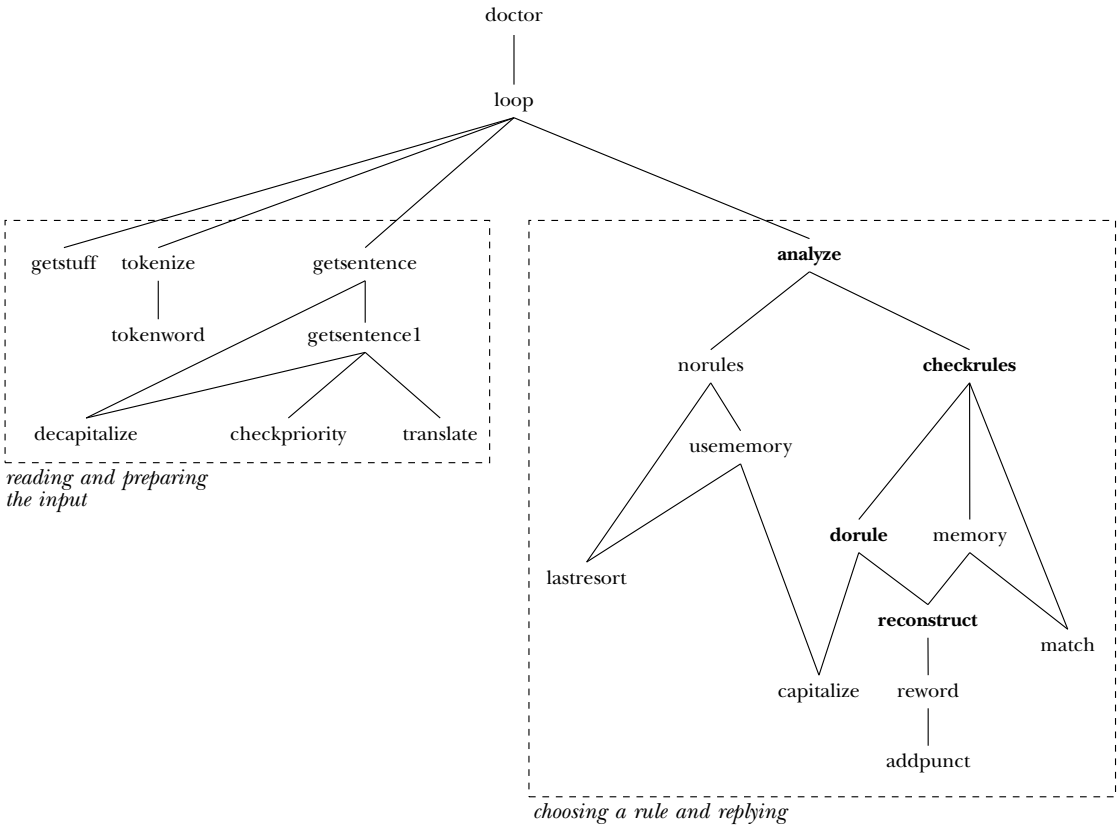
These steps are not really quite sequential; for example, the first two are done in parallel because, if there is more than one sentence, the program chooses a sentence that contains one or more keywords.

I have omitted from this list several possible complications. One important one is that there is a second kind of rule (besides the response rules) called a *memory* rule. These rules do not generate immediate responses to what the user types. Instead, they generate sentences that are appropriate for later use, like "Earlier you said your spirits are low" in the sample dialogue earlier. That response was not generated from the user's comment just before it, but was instead added to memory at the time of the user's first remark. Because the user's comment "Of course" contained no keywords, the program chose to use a response from memory at that time.

Other complications have to do with the nature of the rules associated with a keyword. Instead of patterns and responses, a keyword can just have *another* keyword as its rules, meaning that the rules for that keyword should be used for this one also. Alternatively, another keyword can be used in place of a response for a particular pattern,

so the alternate keyword is used only if that pattern is matched. Another alternative to a response is the word "newkey," which means that the program should abandon this keyword and try another one in the same sentence. Yet another alternative is an instruction to rearrange the sentence and try matching patterns again with the new version. As you read the procedures shown below, try not to get caught up in these complications the first time through. Just forget about the `if` instructions that check for these special cases and concentrate on the usual situation.

I'll explain how each part of the linguistic strategy is carried out by the procedures in the project. Here is a diagram of the subprocedure/superprocedure relationships. The top-level `doctor` does some initialization and then invokes `loop`, which does the real work.



```
to doctor
local [text sentence stuff a b c rules keywords memory]
make "memory []
print [Hello, I am the doctor. What can I do for you?]
print [Please end your remarks with an empty line.]
print []
loop
end

to loop
make "text tokenize getstuff []
make "sentence getsentence :text
analyze :sentence :keywords
print []
loop
end
```

`Loop` uses `getstuff` to read several lines of the user's typing into a long list. This list is processed by several procedures in turn.

```
to getstuff :stuff
localmake "line readlist
if emptyp :line [output :stuff]
output getstuff sentence :stuff :line
end
```

The first step in my numbered list is to find the punctuation and use it to extract a single sentence from the list. The first part, finding the punctuation, is the job of `tokenize` and its subprocedure `tokenword`. (These procedures get their name from the computer scientist's term *token*, which means the smallest possible meaningful string of characters. In the BASIC compiler of Chapter 6, the procedure called `reader` separates a line of text into tokens.)

```
to tokenize :text
output map.se [tokenword ? "] :text
end

to tokenword :word :out
if emptyp :word [output :out]
if memberp first :word [, " ] [output tokenword butfirst :word :out]
if memberp first :word [. ? ! |;|] [output sentence :out ".]
output tokenword butfirst :word word :out first :word
end
```

The program's understanding of punctuation is very simple. Some punctuation, like a quotation mark, is just ignored completely. Periods, question marks, and semicolons are all treated as having the same meaning, namely, they mark the end of a sentence. `tokenword` turns a word like `why?` into the two-word list `[why .]` so that the period as a separate word serves as the separator between sentences in the long list.

Extracting a single sentence is done at the same time as steps 2 and 3, finding keywords and applying translations. All of these are done by `getsentence`, which uses `checkpriority` and `translate` as subprocedures for tasks 2 and 3 respectively. (The version of Doctor in the first edition worked with all capital letters. In this new version, I've added a procedure `decapitalize` that turns the first letter of each sentence to lower case, and when the program prints a response it uses the inverse procedure `capitalize` to capitalize the first word. This is necessary because the first word of the user's sentence might end up in the middle of the program's response, and vice versa.)

```
to getsentence :text
make "keywords []
output getsentence1 decapitalize :text []
end

to getsentence1 :text :out
if emptyp :text [output :out]
if equalp first :text ". ~
   [ifelse emptyp :keywords
           [output getsentence1 decapitalize butfirst :text []]
           [output :out]]
checkpriority first :text
output getsentence1 butfirst :text sentence :out translate first :text
end

to decapitalize :text
if emptyp :text [output []]
output fput lowercase first :text butfirst :text
end

to checkpriority :word
localmake "priority gprop :word "priority
if emptyp :priority [stop]
if emptyp :keywords [make "keywords ( list :word ) stop]
ifelse :priority > ( gprop first :keywords "priority ) ~
       [make "keywords fput :word :keywords] ~
       [make "keywords lput :word :keywords]
end
```

```
to translate :word
localmake "new gprop :word "translation
output ifelse emptyp :new [:word] [:new]
end
```

At each sentence separator (a word containing only a period), `getsentence1` checks whether any keywords have been found yet. If so, the sentence before the separator is the one the program uses. If not, `getsentence1` goes on to examine the next sentence in the list. If the last sentence ends without any keywords found, that last sentence is chosen anyway.

Both `checkpriority` and `translate` work through the use of property lists that are associated with words. The script part of `doctor` (Weizenbaum's second tier) consists of these property lists. For example, here's part of the script setup:

```
pprop "my "priority 2
pprop "my "translation "your
```

The first of these means that `my` is a keyword, with priority 2. In the Doctor script, most keywords have priority 0. `My` is a little more important than most words, but not as important as `dreamed` (priority 4) or `computer` (priority 50)! `Checkpriority` arranges the list of keywords so that the word with highest priority is first in the list. The `translation` property means that the word `my` is changed to `your` in generating responses. For example, at the beginning of the sample dialogue above, the sentence "My spirits are low" is echoed as "Your spirits are low?" (The keyword list created by `checkpriority` has *untranslated* keywords. That's why the patterns associated with the keyword `you`, for example, all contain the word `I` instead; the patterns deal with the translated version.)

Step 4, finding the highest priority keyword, is simply a matter of choosing the `first` keyword in the list because of the way `checkpriority` has done its job. This selection is made by `analyze`, which then invokes `checkrules` as a subprocedure. (`Analyze` also recognizes the special situation in which one keyword refers to the rules of another.)

```
to analyze :sentence :keywords
local [rules keyword]
if emptyp :keywords [norules stop]
make "keyword first :keywords
make "rules gprop :keyword "rules
if wordp first :rules ~
   [make "keyword first :rules make "rules gprop :keyword "rules]
checkrules :keyword :rules
end
```

`Checkrules` handles step 5, finding an applicable rule for the chosen keyword. That is, `checkrules` invokes `match` to match the selected sentence against each pattern associated with the given keyword. (The program assumes that the script is written so that there will always be at least one matching pattern. Most keywords have `[#]` as the pattern in the last rule.) When `checkrules` finds a matching pattern, it invokes `dorule` to examine the corresponding list of responses. (One complication in understanding these procedures is that the input to `dorule` is *the name of a property* whose value is the list of responses. I'll get back to discussing why it's done this way later; for now, all that's really important is that `dorule` chooses one of the responses and uses it as the input to `reconstruct`.)

```
to checkrules :keyword :rules
if not match first :rules :sentence ~
   [checkrules :keyword butfirst butfirst :rules stop]
dorule first butfirst :rules
end

to dorule :rule
localmake "print first gprop :keyword :rule
pprop :keyword :rule lput :print butfirst gprop :keyword :rule
if equalp :print "newkey [analyze :sentence butfirst :keywords stop]
if wordp :print [checkrules :print gprop :print "rules stop]
if equalp first :print "pre ~
   [analyze reconstruct first butfirst :print
            butfirst butfirst :print
    stop]
print capitalize reconstruct :print
memory :keyword :sentence
end
```

The usual task of `dorule` is to carry out step 6, the generation of a response. For example, when the user typed

```
My spirits are low.
```

the highest priority keyword found was `my`. There are three patterns associated with this keyword:

```
[# your # !a:familyp #b]
[# your &stuff]
[#]
```

(Again, the pattern is matched against the sentence *after* translation, so the patterns contain the word `your` even though the actual keyword is `my`.) The first of these patterns does not match the sentence. (`Familyp` is a predicate that's true for words like `mother` and `brother`.) The second pattern, however, does match the sentence. `Match` gives the variable `stuff` the list

```
[spirits are low]
```

as its value. (The period that ended what the user typed was removed by `tokenize`.)

Associated with that second pattern is this list of responses:

```
[[Your :stuff?]
 [Why do you say your :stuff?]
 [Does that suggest anything else which belongs to you?]
 [Is it important to you that your :stuff?]]
```

`Dorule` chooses the first of these, and invokes `reconstruct` to substitute the actual value of the variable into the response. By the way, although I've used Logo's notation of colon to mean "the value of the variable," `reconstruct` isn't exactly like the Logo interpreter. For one thing, it recognizes punctuation marks; it knows that this response refers to a variable named `stuff`, not `stuff?`.

```
to reconstruct :sentence
if emptyp :sentence [output []]
if not equalp ": first first :sentence ~
   [output fput first :sentence reconstruct butfirst :sentence]
output sentence reword first :sentence reconstruct butfirst :sentence
end

to reword :word
if memberp last :word [. ? ,] ~
   [output addpunct reword butlast :word last :word]
output thing butfirst :word
end

to addpunct :stuff :char
if wordp :stuff [output word :stuff :char]
if emptyp :stuff [output :char]
output sentence butlast :stuff word last :stuff :char
end
```

Stimulus-Response Psychology

Historically, there have been two ways of looking at the purpose of artificial intelligence research. One way is to see it as research into what computers can do, and into the meaning of intelligence in general, without any special reference to how *people* think. Researchers who take this approach are willing to use any technique that will solve a problem, even if it's perfectly obvious that people don't think that way. The second approach is to see artificial intelligence as a way to shed light on human intelligence. In this approach, the idea is to use the computer as a *model* for the human mind. Researchers who follow this path try to write their programs to mimic human behavior and, they hope, even the inner mechanisms of human brains. Recently there has been a tendency for researchers to declare themselves as wholly in one camp or the other. People who want to solve problems even if by non-human methods are part of the "knowledge engineering" field; the programs they develop are called "expert systems." People who want to use the computer to help build theories of human intelligence are in the field of "cognitive science."

Weizenbaum's work on Eliza is an early example of the former approach. He was emphatically *not* claiming that his program worked the way people work. Indeed, one of the purposes of the program was to demonstrate how realistic the *behavior* of a computer program can be, even when we are quite sure that the underlying *mechanism* is completely unrealistic.

Nevertheless, Eliza could be taken as a computer model of a certain theory of human psychology. It may not be obvious what it means for a computer program to model a theory about people, so it may be worthwhile to examine Eliza from this point of view. One theory about how people think is called *behaviorism,* or the *stimulus-response* theory. According to this theory, a person's mind is a "black box," and we can't know what's inside. What we *can* know, however, is how a person reacts to different situations and events. Whatever situation presents itself to you is called a *stimulus.* A stimulus can be something very abrupt like an electric shock, or it can be something more subtle like a particular sentence spoken by a particular person. When you are presented with a given stimulus, you produce a certain *response*: you say something back, or you jump, or you fall asleep. People learn to associate certain responses with certain stimuli. People can be trained to change the response associated with a stimulus by using *conditioning* techniques. If you are rewarded for producing a certain response, you'll produce it more often.

The behaviorist theory was very influential several years ago, although hardly anyone believes it any more. What would it mean to write a computer model for this theory? Well, the model would have two main parts: one that recognizes stimuli, and one that produces

a response for a given stimulus. In Eliza, the first part is the pattern matcher. I haven't spoken much about that part of the program in this description because I discussed it at length in the last project. But in fact `match` and its subprocedures are a substantial part of the complete `doctor` program. The second part, the one that produces responses, is `dorule` and `reconstruct`.

Eliza does not represent a very sophisticated form of stimulus-response theory because it leaves out the idea of *learning*. In Eliza, the responses are all provided in advance, as part of the script. People can develop new responses over time, and behaviorist theory has a lot to say about exactly what the rules are that govern such learning. (That's what education is, to a behaviorist: learning new responses to stimuli.) Since the script for Eliza is stored in lists, and those lists can be manipulated by the program, it would be possible to modify the rules of the program so that it can learn new rules while it's running. For example, if the user types something like "What are you talking about?" then the program could decide that its previous response was inappropriate. It would learn to avoid that response next time. You might like to think about how to design such an extension to the project as presented here.

Researchers who, unlike Weizenbaum, are deliberately trying to model theories of human psychology generally use a more complicated program structure. For example, experiments measuring the reaction time of human beings in different situations seem to indicate that people have a short-term memory and a long-term memory. The former may hold the telephone number you're dialing right now, for instance, while the latter holds all the telephone numbers of all your friends. Short-term memory is faster than long-term, but much smaller; you can only remember a few things at a time in it. Computers do not inherently have these two kinds of memory; they're really good at remembering many things *and* finding them quickly. But cognitive scientists write programs that deliberately limit the computer's ability to remember things quickly, trying to model the inner structure of the brain in this way.

Property Lists

The response rules, memory rules, translations, and priorities that make up the script are all stored in the form of property lists. Each keyword has a property list. For example, the property list for the word `my` looks like this:

```
[priority 2
 translation your
 rules [[# your # !a:familyp #b] g1 [# your &stuff] g2 [#] g3]
```

```
 g1 [[Tell me more about your family.]
     [Who else in your family :b?]
     [Your :a?]
     [What else comes to mind when you think of your :a?]]
 g2 [[Your :stuff?]
     [Why do you say your :stuff?]
     [Does that suggest anything else which belongs to you?]
     [Is it important to you that your :stuff?]]
 g3 [newkey]
 memr [[# your &stuff] g4]
 g4 [[Earlier you said your :stuff.]
     [But your :stuff.]
     [Does that have anything to do with your statement about :stuff?]]]
```

Remember that a property list contains pairs of members; the odd numbered members are names of properties, and the even numbered members are the values of those properties. The word `my` has properties named `priority`, `translation`, `rules`, `g1`, `g2`, `g3`, `memr`, and `g4`. The `priority` and `translation` properties are straightforward. The `rules` and `memr` properties have as their values lists in which the odd numbered members are patterns and the even numbered members are names of other properties. These other properties contain the lists of responses. The names of these secondary properties are arbitrary and are generated by the program.

To create these property lists, I used `pprop` directly for some of the properties, but wrote setup procedures to help with the more complicated parts. Here are the instructions that contribute to this property list.

```
pprop "my "priority 2
pprop "my "translation "your
addrule "my [# your # !a:familyp #b]
  [[Tell me more about your family.]
   [Who else in your family :b?]
   [Your :a?]
   [What else comes to mind when you think of your :a?]]
addrule "my [# your &stuff]
  [[Your :stuff?]
   [Why do you say your :stuff?]
   [Does that suggest anything else which belongs to you?]
   [Is it important to you that your :stuff?]]
addrule "my [#] [newkey]
addmemr "my [# your &stuff]
  [[Earlier you said your :stuff.]
   [But your :stuff.]
   [Does that have anything to do with your statement about :stuff?]]
```

In general, the order in which properties are added to the list doesn't matter. However, the order of the `addrule` instructions *does* matter, because the rule that's added first is the one that `checkrules` tries first. It's important, therefore, that the rules go from most specific pattern to least specific pattern. In this case, the first pattern checks for a remark about a member of the user's family; the second checks for a remark about some other object or characteristic belonging to the user; and the third is a catch-all pattern just in case the other two fail.

Generated Symbols

The procedures `addrule` and `addmemr` are very similar, since the `rules` and `memr` properties are similar in format.

```
to addrule :word :pattern :results
localmake "propname gensym
pprop :word "rules (sentence gprop :word "rules list :pattern :propname)
pprop :word :propname :results
end

to addmemr :word :pattern :results
localmake "propname gensym
pprop :word "memr (sentence gprop :word "memr list :pattern :propname)
pprop :word :propname :results
end
```

Each of these procedures uses a local variable `propname` to contain the name of the response property, a "generated symbol" or *gensym*. These are the words like `g3` in the example above. Each procedure carries out two `pprop` instructions. The first appends a new pattern and a new gensym to the previous value of the `rules` or `memr` property; the second creates a new property with the gensym as its name and the response (or memory) list as its value. `Gensym` is a Berkeley Logo library procedure.

Modification of List Structure

Why are generated symbols needed in this program at all? In the Lisp version of Doctor, property lists are still used, but the entire collection of rules is one big list, the value of the property `rules`. It's as if the Logo property list looked like this:

```
[priority 2
 translation your
 rules
   [[# your # !a:familyp #b]
      [[Tell me more about your family.]
       [Who else in your family :b?]
       [Your :a?]
       [What else comes to mind when you think of your :a?]]
    [# your &stuff]
      [[Your :stuff?]
       [Why do you say your :stuff?]
       [Does that suggest anything else which belongs to you?]
       [Is it important to you that your :stuff?]]
    [#]
      [newkey]]
 memr
   [[# your &stuff]
      [[Earlier you said your :stuff.]
       [But your :stuff.]
       [Does that have anything to do with your statement
        about :stuff?]]]
]
```

I chose not to use one big list of rules in the Logo version. In Lisp (and in Berkeley Logo, but not in the versions of Logo I had available when writing the first edition), it's possible to change one of the members of a list without recopying the rest of the list. Without that capability, it's better to divide the rules into separate, smaller lists, so that only a little recopying is needed to change one.

Each pattern has several responses associated with it. When the program matches a particular pattern, it does not choose a response at random. Instead, it rotates through the list of responses in order. That is, it uses the first response first, then the second, and so on until the end of the list; if another response is needed, it starts over at the beginning of the list. This strict rotation is sometimes important because some of the responses say things like "I already told you that..."

The way the program keeps track of the rotation of the responses for a given rule is that it actually changes the response list so that what used to be the first response is moved to the end. Thus, `dorule` contains the instructions

```
localmake "print first gprop :keyword :rule
pprop :keyword :rule lput :print butfirst gprop :keyword :rule
```

The first of these instructions extracts the first response from the list of responses. The second one replaces the list of responses with a new list, in which the old first response is `lput` behind the remaining ones.

What if the rules were one big list? To see what would be required, let's look at a smaller list, in which it will be easier to follow what needs to be changed. Suppose some word's `rules` property had as its value this list:

```
[1 [A B C] 2 [D E F] 3 [G H I]]
```

In this example, the numbers represent patterns, while the letters represent responses. Now suppose that the program finds a match for pattern number `2`. It should then issue the response `D`. Then it should rotate the three responses associated with pattern `2` so that the new `rules` property is

```
[1 [A B C] 2 [E F D] 3 [G H I]]
```

The only way to do this in most versions of Logo is to construct a new copy of the entire list. Here is a way you could write such a program:

```
to rotate :keyword :pattern
pprop :keyword "rules (rotate1 :pattern gprop :keyword "rules)
end

to rotate1 :pattern :rules
if emptyp :rules [output []]
if equalp :pattern first :rules
   [output sentence (list :pattern rotate2 first butfirst :rules)
                    (butfirst butfirst :rules)]
output sentence (list first :rules first butfirst :rules) ~
                (rotate1 :pattern butfirst butfirst :rules)
end

to rotate2 :list
output lput first :list butfirst :list
end
```

You'd use this program with an instruction like

```
rotate "word 2
```

where `2` represents the pattern in the example above.

The trouble with this approach is that it's slow. It does a lot of `list` and `sentence` operations to reconstruct the modified list. More importantly, the *entire* list must be copied, even though only one rule is to be modified.

In Lisp, and in Berkeley Logo, there are primitive commands that can be used to change the contents of a list without recopying the unchanged parts. In Berkeley Logo they are called `.setfirst` and `.setbutfirst`; using them, we could write `rotate` this way:

```
to rotate :keyword :pattern
rotate1 :pattern (gprop :keyword "rules)
end

to rotate1 :pattern :rules
if emptyp :rules [stop]
ifelse equalp :pattern first :rules ~
       [.setfirst (butfirst :rules) (rotate2 first butfirst :rules)]
       [rotate1 :pattern butfirst butfirst :rules]
end
```

(I'll leave `rotate2` the same as in the earlier version, for now.) This is a tricky sort of procedure. Here's a trace of how it might be used:

```
rotate "word 2
  rotate1 2 [1 [A B C] 2 [D E F] 3 [G H I]]
    rotate1 2 [2 [D E F] 3 [G H I]]
```

In the lower-level invocation of `rotate1`, the `equalp` test outputs `true`, so the `ifelse` instruction evaluates its second input. This is equivalent to the instruction

```
.setfirst [[D E F] 3 [G H I]] (rotate2 [D E F])
```

or

```
.setfirst [[D E F] 3 [G H I]] [E F D]
```

To understand what this means, you must realize that the primitive operation `butfirst` does not make a *copy* of the butfirst of its input. Instead, the list output by `butfirst` is actually part of the list that is its input—they share the same cells in the computer's memory. Therefore, to change something in the `butfirst` also changes the larger list. The `setfirst` instruction ends up changing the `rules` property of the word `word` even though there is no explicit `pprop` to change the property.

If you're not a Lisp programmer, this probably seems like magic. It certainly violates some of the rules you've learned about the evaluation of Logo instructions. For example, if you actually typed the instruction

```
.setfirst [[D E F] 3 [G H I]] [E F D]
```

explicitly at top level, it would *not* change the property list of `word`, because the list that `setfirst` modifies would *not* be part of that property list, even though it has the same members. It's only because `setfirst`'s input is derived from that property list by a series of `butfirst` operations that they share the same memory.

Do you find this confusing? The original designers of Logo chose not to include `.setfirst` in the language because it *is* hard to understand, and because it can produce some very strange results if you're not careful with it. For example, consider these instructions:

```
make "c [x y]
.setfirst (butfirst :c) (:c)
```

This `.setfirst` instruction will produce a *circular list*, one that contains itself as a member. If you try to print `:c`, you'll see something like

```
[x [x [x [x [x [x [x [x [x [x [x [x [x [x ...
```

going on forever.

Once we have these list modification tools, even the implicit recopying done by `lput` can be avoided. Here's a more efficient version of `rotate1`, but it's really tricky to understand and it isn't a technique that I recommend:

```
to rotate1 :pattern :rules
if emptyp :rules [stop]
ifelse equalp :pattern first :rules ~
       [rotate2 butfirst :rules]
       [rotate1 :pattern butfirst butfirst :rules]
end

to rotate2 :rulelist
localmake "firstresponse first :rulelist
localmake "restresponses butfirst :firstresponse
.setfirst :rulelist :restresponses
.setbutfirst :firstresponse []
while [not emptyp butfirst :restresponses] ~
      [make "restresponses butfirst :restresponses]
.setbutfirst :restresponses :firstresponse
end
```

In `rotate2`, the `.setfirst` instruction removes the first response from the head of the list of responses; the two `.setbutfirst` instructions "splice" that first response back into the list at the end, following what used to be the last response.

Leaving `.setfirst` out of Logo was a controversial decision. Some people take the position that, as a "language for learners," Logo should not include mechanisms for which we can't provide an easy-to-follow metaphor; it's counterproductive for the language to encourage you to think in terms of what's where in memory. Other people refer to this idea scornfully as "protecting the user from himself," arguing that if a mechanism is useful it should be provided even though it's error-prone.

In any case, since Logo didn't have `.setfirst` and I didn't want the `doctor` program to be slowed down by having to recopy the `rules` property all the time, I decided to make each response list a separate property, so that each response list can be modified independently of the others. That's the reason for the gensym property names: so that `dorule` can rotate the responses for a particular pattern without disturbing the responses for other patterns. I could have changed this in the Berkeley Logo version, but it didn't seem worthwhile; using names for the rules is a little inelegant but doesn't hurt the program's efficiency.

Linguistic Structure

Because it treats a sentence as simply a string of words, Eliza is limited in its linguistic sophistication. For example, the Doctor script has this pattern associated with the keyword `I`:

```
[# you are # !stuff:in [sad unhappy depressed sick] #]
```

(Remember that the pattern is matched against the input sentence after translation, so the words `you are` in the pattern really match a sentence containing the words `I am`.) The purpose of the pattern is to match a sentence like

```
I think I am really depressed because Susan doesn't like me.
```

The response of the program might be

```
I'm sorry to hear you are depressed.
```

The `#` between `are` and `!stuff` in the pattern is meant to catch adverbs like the word `really` in the example I just gave. But it could also "absorb" some more structurally important parts of a sentence:

```
I am sure that I would be depressed if she left me.
```

This sentence matches the pattern, but it doesn't really fit the intent of the pattern. The person who types this sentence is not saying "I am depressed" at all.

The trouble is that the string of words "sure that I would be" is not equivalent to an adverb. In fact, these words do not form a phrase at all. The program is making a grammatical error by connecting the word `depressed` with the word `am` as a predicate adjective. To avoid such errors, it's not good enough to have more and more detailed patterns to match. You can't anticipate every possible string of words by that technique. Instead, the program would have to impose a tree structure on the sentence, sort of like what you did in diagraming sentences in elementary school. The true structure of this sentence is something like

```
[[subject I] [predicate [[verb am] [nominative [[adjective sure]
   [adverb [clause [[connective that] [subject I] [verb [would be] ...
```

and so on. (Actually, I've just made up this structure to illustrate the idea, and it's not very realistic. I've tried too hard to preserve the order of the words in the original sentence. A more practical structure would probably center on the verb in each clause, and have subordinate slots for the subject, object, and so on. A connective like "that" might just be thrown away completely; the purpose served by such words in spoken text would instead be filled by the very sublist organization itself.) People have in fact written several computer programs that transform English sentences into a structured representation. It's very hard to do a perfect job, though, because of problems like homonyms: the word "like" can be a verb (I like ice cream) or a preposition (I want to be like my big brother).

Further Explorations

There are three main directions in which you can explore the territory that this project begins. First, you can try to refine the existing Doctor script, so that it does a better job within the same general framework. Whenever you get an inappropriate response from `doctor`, see if you can think of a new rule that would solve that case without messing up other sentences.

A second possibility would be to write an entirely new Eliza script, so that instead of being a doctor the program can carry on some different sort of conversation. How about taking orders in a fast food restaurant? Answering questions from some data base about presidents or baseball players?

The third direction would be to abandon Eliza and look into some of the other approaches to understanding and generating English sentences that have been developed.

Program Listing

The procedures from the pattern matcher of Chapter 7 are included in this program, but they are not listed again here.

```
to doctor
local [text sentence stuff a b c rules keywords memory]
make "memory []
print [Hello, I am the doctor. What can I do for you?]
print [Please end your remarks with an empty line.]
print []
loop
end

;; Controlling the conversation

to loop
make "text tokenize getstuff []
make "sentence getsentence :text
analyze :sentence :keywords
print []
loop
end

;; Reading and preparing the input

to getstuff :stuff
localmake "line readlist
if emptyp :line [output :stuff]
output getstuff sentence :stuff :line
end

to tokenize :text
output map.se [tokenword ? "] :text
end

to tokenword :word :out
if emptyp :word [output :out]
if memberp first :word [, " ] [output tokenword butfirst :word :out]
if memberp first :word [. ? ! |;|] [output sentence :out ".]
output tokenword butfirst :word word :out first :word
end

to getsentence :text
make "keywords []
output getsentence1 decapitalize :text []
end
```

```
to getsentence1 :text :out
if emptyp :text [output :out]
if equalp first :text ". ~
   [ifelse emptyp :keywords ~
           [output getsentence1 decapitalize butfirst :text []] [output :out]]
checkpriority first :text
output getsentence1 butfirst :text sentence :out translate first :text
end

to decapitalize :text
if emptyp :text [output []]
output fput lowercase first :text butfirst :text
end

to checkpriority :word
localmake "priority gprop :word "priority
if emptyp :priority [stop]
if emptyp :keywords [make "keywords ( list :word ) stop]
ifelse :priority > ( gprop first :keywords "priority ) ~
       [make "keywords fput :word :keywords] ~
       [make "keywords lput :word :keywords]
end

to translate :word
localmake "new gprop :word "translation
output ifelse emptyp :new [:word] [:new]
end

;; Choosing the rule and replying

to analyze :sentence :keywords
local [rules keyword]
if emptyp :keywords [norules stop]
make "keyword first :keywords
make "rules gprop :keyword "rules
if wordp first :rules ~
   [make "keyword first :rules make "rules gprop :keyword "rules]
checkrules :keyword :rules
end

to checkrules :keyword :rules
if not match first :rules :sentence ~
   [checkrules :keyword butfirst butfirst :rules stop]
dorule first butfirst :rules
end
```

```
to dorule :rule
localmake "print first gprop :keyword :rule
pprop :keyword :rule lput :print butfirst gprop :keyword :rule
if equalp :print "newkey [analyze :sentence butfirst :keywords stop]
if wordp :print [checkrules :print gprop :print "rules stop]
if equalp first :print "pre ~
   [analyze reconstruct first butfirst :print butfirst butfirst :print stop]
print capitalize reconstruct :print
memory :keyword :sentence
end

to reconstruct :sentence
if emptyp :sentence [output []]
if not equalp ": first first :sentence ~
   [output fput first :sentence reconstruct butfirst :sentence]
output sentence reword first :sentence reconstruct butfirst :sentence
end

to reword :word
if memberp last :word [. ? ,] [output addpunct reword butlast :word last :word]
output thing butfirst :word
end

to addpunct :stuff :char
if wordp :stuff [output word :stuff :char]
if emptyp :stuff [output :char]
output sentence butlast :stuff word last :stuff :char
end

to capitalize :text
if emptyp :text [output []]
output fput (word uppercase first first :text butfirst first :text) butfirst :text
end

to memory :keyword :sentence
local [rules rule name]
make "rules gprop :keyword "memr
if emptyp :rules [stop]
if not match first :rules :sentence [stop]
make "name last :rules
make "rules gprop :keyword :name
make "rule first :rules
pprop :keyword :name lput :rule butfirst :rules
make "memory fput reconstruct :sentence :memory
end

to norules
ifelse :memflag [usememory] [lastresort]
make "memflag not :memflag
end
```

```
to lastresort
print first :lastresort
make "lastresort lput first :lastresort butfirst :lastresort
end

to usememory
if emptyp :memory [lastresort stop]
print capitalize first :memory
make "memory butfirst :memory
end

;; Predicates for patterns

to beliefp :word
output not emptyp gprop :word "belief
end

to familyp :word
output not emptyp gprop :word "family
end

;; Procedures for adding to the script

to addrule :word :pattern :results
localmake "propname gensym
pprop :word "rules (sentence gprop :word "rules list :pattern :propname)
pprop :word :propname :results
end

to addmemr :word :pattern :results
localmake "propname gensym
pprop :word "memr (sentence gprop :word "memr list :pattern :propname)
pprop :word :propname :results
end

;; data

make "gensym.number 80

make "lastresort [[I am not sure I understand you fully.] [Please go on.]
                  [What does that suggest to you?]
                  [Do you feel strongly about discussing such things?]]

make "memflag "false

pprop "alike "priority 10
pprop "alike "rules [dit]
```

```
pprop "always "priority 1
pprop "always "rules [[#] g69]
pprop "always "g69 [[Can you think of a specific example?] [When?]
                    [What incident are you thinking of?]
                    [Really, always?] [What if this never happened?]]

pprop "am "priority 0
pprop "am "translation "are
pprop "am "rules [[# are you #stuff] g18 [#] g19]
pprop "am "g18 [[Do you believe you are :stuff?] [Would you want to be :stuff?]
                [You wish I would tell you you are :stuff.]
                [What would it mean if you were :stuff?] how]
pprop "am "g19 [[Why do you say "am"?] [I don't understand that.]]

pprop "are "priority 0
pprop "are "rules [[#a there are #b you #c] g20 [# there are &stuff] g21
                   [# are I #stuff] g22 [are #] g23 [# are #stuff] g24]
pprop "are "g20 [[pre [:a there are :b] are]]
pprop "are "g21 [[What makes you think there are :stuff?]
                 [Do you usually consider :stuff?]
                 [Do you wish there were :stuff?]]
pprop "are "g22 [[Why are you interested in whether I am :stuff or not?]
                 [Would you prefer if I weren't :stuff?]
                 [Perhaps I am :stuff in your fantasies.]
                 [Do you sometimes think I am :stuff?] how]
pprop "are "g23 [how]
pprop "are "g24 [[Did you think they might not be :stuff?]
                 [Would you like it if they were not :stuff?]
                 [What if they were not :stuff?] [Possibly they are :stuff.]]

pprop "ask "priority 0
pprop "ask "rules [[# you ask #] g77 [# you ! asking #] g78 [# I #] g79 [#] g80]
pprop "ask "g77 [how]
pprop "ask "g78 [how]
pprop "ask "g79 [you]
pprop "ask "g80 [newkey]

pprop "because "priority 0
pprop "because "rules [[#] g64]
pprop "because "g64 [[Is that the real reason?]
                     [Don't any other reasons come to mind?]
                     [Does that reason seem to explain anything else?]
                     [What other reasons might there be?]
                     [You're not concealing anything from me, are you?]]

pprop "believe "belief "true

pprop "bet "belief "true

pprop "brother "family "true
```

```
pprop "can "priority 0
pprop "can "rules [[# can I #stuff] g58 [# can you #stuff] g59 [#] g60]
pprop "can "g58 [[You believe I can :stuff, don't you?] how
                [You want me to be able to :stuff.]
                [Perhaps you would like to be able to :stuff yourself.]]
pprop "can "g59 [[Whether or not you can :stuff depends more on you than on me.]
                [Do you want to be able to :stuff?]
                [Perhaps you don't want to :stuff.] how]
pprop "can "g60 [how newkey]

pprop "cant "translation "can't

pprop "certainly "priority 0
pprop "certainly "rules [yes]

pprop "children "family "true

pprop "computer "priority 50
pprop "computer "rules [[#] g17]
pprop "computer "g17 [[Do computers worry you?]
                      [Why do you mention computers?]
                      [What do you think machines have to do with your problem?]
                      [Don't you think computers can help people?]
                      [What about machines worries you?]
                      [What do you think about machines?]]

pprop "computers "priority 50
pprop "computers "rules [computer]

pprop "dad "translation "father
pprop "dad "family "true

pprop "daddy "translation "father
pprop "daddy "family "true

pprop "deutsch "priority 0
pprop "deutsch "rules [[#] g15]
pprop "deutsch "g15 [[I'm sorry, I speak only English.]]

pprop "dit "rules [[#] g72]
pprop "dit "g72 [[In what way?] [What resemblance do you see?]
                [What does that similarity suggest to you?]
                [What other connections do you see?]
                [What do you suppose that resemblance means?]
                [What is the connection, do you suppose?]
                [Could there really be some connection?] how]

pprop "dont "translation "don't
```

```
pprop "dream "priority 3
pprop "dream "rules [[#] g9]
pprop "dream "g9 [[What does that dream suggest to you?] [Do you dream often?]
                 [What persons appear in your dreams?]
                 [Don't you believe that dream has something to do
                  with your problem?]
                 [Do you ever wish you could flee from reality?] newkey]

pprop "dreamed "priority 4
pprop "dreamed "rules [[# you dreamed #stuff] g7 [#] g8]
pprop "dreamed "g7 [[Really :stuff?]
                    [Have you ever fantasied :stuff while you were awake?]
                    [Have you dreamed :stuff before?] dream newkey]
pprop "dreamed "g8 [dream newkey]

pprop "dreams "translation "dream
pprop "dreams "priority 3
pprop "dreams "rules [dream]

pprop "dreamt "translation "dreamed
pprop "dreamt "priority 4
pprop "dreamt "rules [dreamed]

pprop "espanol "priority 0
pprop "espanol "rules [deutsch]

pprop "everybody "priority 2
pprop "everybody "rules [everyone]

pprop "everyone "priority 2
pprop "everyone "rules [[# !a:in [everyone everybody nobody noone] #] g68]
pprop "everyone "g68 [[Really, :a?] [Surely not :a.]
                      [Can you think of anyone in particular?]
                      [Who, for example?]
                      [You are thinking of a very special person.]
                      [Who, may I ask?] [Someone special perhaps.]
                      [You have a particular person in mind, don't you?]
                      [Who do you think you're talking about?]
                      [I suspect you're exaggerating a little.]]

pprop "father "family "true

pprop "feel "belief "true

pprop "francais "priority 0
pprop "francais "rules [deutsch]

pprop "hello "priority 0
pprop "hello "rules [[#] g16]
pprop "hello "g16 [[How do you do. Please state your problem.]]
```

```
pprop "how "priority 0
pprop "how "rules [[#] g63]
pprop "how "g63 [[Why do you ask?] [Does that question interest you?]
                 [What is it you really want to know?]
                 [Are such questions much on your mind?]
                 [What answer would please you most?] [What do you think?]
                 [What comes to your mind when you ask that?]
                 [Have you asked such questions before?]
                 [Have you asked anyone else?]]

pprop "husband "family "true

pprop "i "priority 0
pprop "i "translation "you
pprop "i "rules [[# you !:in [want need] #stuff] g32
                 [# you are # !stuff:in [sad unhappy depressed sick] #] g33
                 [# you are # !stuff:in [happy elated glad better] #] g34
                 [# you was #] g35 [# you !:beliefp you #stuff] g36
                 [# you # !:beliefp # i #] g37 [# you are #stuff] g38
                 [# you !:in [can't cannot] #stuff] g39
                 [# you don't #stuff] g40 [# you feel #stuff] g41
                 [# you #stuff i #] g42 [#stuff] g43]
pprop "i "g32 [[What would it mean to you if you got :stuff?]
               [Why Do you want :stuff?] [Suppose you got :stuff soon.]
               [What if you never got :stuff?]
               [What would getting :stuff mean to you?] [You really want :stuff.]
               [I suspect you really don't want :stuff.]]
pprop "i "g33 [[I'm sorry to hear you are :stuff.]
               [Do you think coming here will help you not to be :stuff?]
               [I'm sure it's not pleasant to be :stuff.]
               [Can you explain what made you :stuff?] [Please go on.]]
pprop "i "g34 [[How have I helped you to be :stuff?]
               [Has your treatment made you :stuff?]
               [What makes you :stuff just now?]
               [Can you explain why you are suddenly :stuff?]
               [Are you sure?] [What do you mean by :stuff?]]
pprop "i "g35 [was]
pprop "i "g36 [[Do you really think so?] [But you are not sure you :stuff.]
               [Do you really doubt you :stuff?]]
pprop "i "g37 [you]
pprop "i "g38 [[Is it because you are :stuff that you came to me?]
               [How long have you been :stuff?]
               [Do you believe it normal to be :stuff?]
               [Do you enjoy being :stuff?]]
pprop "i "g39 [[How do you know you can't :stuff?] [Have you tried?]
               [Perhaps you could :stuff now.]
               [Do you really want to be able to :stuff?]]
pprop "i "g40 [[Don't you really :stuff?] [Why don't you :stuff?]
               [Do you wish to be able to :stuff?] [Does that trouble you?]]
```

```
pprop "i "g41 [[Tell me more about such feelings.] [Do you often feel :stuff?]
              [Do you enjoy feeling :stuff?]
              [Of what does feeling :stuff remind you?]]
pprop "i "g42 [[Perhaps in your fantasy we :stuff each other.]
              [Do you wish to :stuff me?] [You seem to need to :stuff me.]
              [Do you :stuff anyone else?]]
pprop "i "g43 [[You say :stuff.] [Can you elaborate on that?]
              [Do you say :stuff for some special reason?]
              [That's quite interesting.]]

pprop "i'm "priority 0
pprop "i'm "translation "you're
pprop "i'm "rules [[# you're #stuff] g31]
pprop "i'm "g31 [[pre [you are :stuff] I]]

pprop "if "priority 3
pprop "if "rules [[#a if #b had #c] g5 [# if #stuff] g6]
pprop "if "g5 [[pre [:a if :b might have :c] if]]
pprop "if "g6 [[Do you think it's likely that :stuff?] [Do you wish that :stuff?]
              [What do you think about :stuff?]]

pprop "is "priority 0
pprop "is "rules [[&a is &b] g61 [#] g62]
pprop "is "g61 [[Suppose :a were not :b.] [Perhaps :a really is :b.]
               [Tell me more about :a.]]
pprop "is "g62 [newkey]

pprop "italiano "priority 0
pprop "italiano "rules [deutsch]

pprop "like "priority 10
pprop "like "rules [[# !:in [am is are was] # like #] g70 [#] g71]
pprop "like "g70 [dit]
pprop "like "g71 [newkey]

pprop "machine "priority 50
pprop "machine "rules [computer]

pprop "machines "priority 50
pprop "machines "rules [computer]

pprop "maybe "priority 0
pprop "maybe "rules [perhaps]

pprop "me "translation "you

pprop "mom "translation "mother
pprop "mom "family "true
```

```
pprop "mommy "translation "mother
pprop "mommy "family "true

pprop "mother "family "true

pprop "my "priority 2
pprop "my "translation "your
pprop "my "rules [[# your # !a:familyp #b] g55 [# your &stuff] g56 [#] g57]
pprop "my "g55 [[Tell me more about your family.] [Who else in your family :b?]
                [Your :a?] [What else comes to mind when you think of your :a?]]
pprop "my "g56 [[Your :stuff?] [Why do you say your :stuff?]
                [Does that suggest anything else which belongs to you?]
                [Is it important to you that your :stuff?]]
pprop "my "g57 [newkey]
pprop "my "memr [[# your &stuff] g12]
pprop "my "g12 [[Earlier you said your :stuff.] [But your :stuff.]
                [Does that have anything to do with your statement about :stuff?]]

pprop "myself "translation "yourself

pprop "name "priority 15
pprop "name "rules [[#] g14]
pprop "name "g14 [[I am not interested in names.]
                  [I've told you before I don't care about names\;
                   please continue.]]

pprop "no "priority 0
pprop "no "rules [[no] g53 [#] g54]
pprop "no "g53 [xxyyzz [pre [x no] no]]
pprop "no "g54 [[Are you saying "no" just to be negative?]
                [You are being a bit negative.] [Why not?] [Why "no"?] newkey]

pprop "nobody "priority 2
pprop "nobody "rules [everyone]

pprop "noone "priority 2
pprop "noone "rules [everyone]

pprop "perhaps "priority 0
pprop "perhaps "rules [[#] g13]
pprop "perhaps "g13 [[You don't seem quite certain.] [Why the uncertain tone?]
                     [Can't you be more positive?] [You aren't sure.]
                     [Don't you know?]]
```

```
pprop "problem "priority 5
pprop "problem "rules [[#a !b:in [is are] your !c:in [problem problems] #] g73
                       [# your !a:in [problem problems] !b:in [is are] #c] g74
                       [#] g75]
pprop "problem "g73 [[:a :b your :c.] [Are you sure :a :b your :c?]
                     [Perhaps :a :b not your real :c.]
                     [You think you have problems?]
                     [Do you often think about :a?]]
pprop "problem "g74 [[Your :a :b :c?] [Are you sure your :a :b :c?]
                     [Perhaps your real :a :b not :c.]
                     [You think you have problems?]]
pprop "problem "g75 [[Please continue, this may be interesting.]
                     [Have you any other problems you wish to discuss?]
                     [Perhaps you'd rather change the subject.]
                     [You seem a bit uneasy.] newkey]
pprop "problem "memr [[#stuff is your problem #] g76]
pprop "problem "g76 [[Earlier you mentioned :stuff.]
                     [Let's talk further about :stuff.]
                     [Tell me more about :stuff.]
                     [You haven't mentioned :stuff for a while.]]

pprop "problems "priority 5
pprop "problems "rules [problem]

pprop "remember "priority 5
pprop "remember "rules [[# you remember #stuff] g2
                        [# do I remember #stuff] g3 [#] g4]
pprop "remember "g2 [[Do you often think of :stuff?]
                     [Does thinking of :stuff bring anything else to mind?]
                     [What else do you remember?]
                     [Why do you remember :stuff just now?]
                     [What in the present situation reminds you of :stuff?]]
pprop "remember "g3 [[Did you think I would forget :stuff?]
                     [Why do you think I should recall :stuff now?]
                     [What about :stuff?] what [You mentioned :stuff.]]
pprop "remember "g4 [newkey]

pprop "same "priority 10
pprop "same "rules [dit]

pprop "sister "family "true

pprop "sorry "priority 0
pprop "sorry "rules [[#] g1]
pprop "sorry "g1 [[Please don't apologize.] [Apologies are not necessary.]
                  [What feelings do you have when you apologize?]
                  [I've told you that apologies are not required.]]

pprop "svenska "priority 0
pprop "svenska "rules [deutsch]
```

```
pprop "think "belief "true

pprop "was "priority 2
pprop "was "rules [[# was you #stuff] g26 [# you was #stuff] g27
                   [# was I #stuff] g28 [#] g29]
pprop "was "g26 [[What if you were :stuff?] [Do you think you were :stuff?]
                 [Were you :stuff?] [What would it mean if you were :stuff?]
                 [What does " :stuff " suggest to you?] how]
pprop "was "g27 [[Were you really?] [Why do you tell me you were :stuff now?]
                 [Perhaps I already knew you were :stuff.]]
pprop "was "g28 [[Would you like to believe I was :stuff?]
                 [What suggests that I was :stuff?] [What do you think?]
                 [Perhaps I was :stuff.] [What if I had been :stuff?]]
pprop "was "g29 [newkey]

pprop "we "translation "you
pprop "we "priority 0
pprop "we "rules [I]

pprop "were "priority 0
pprop "were "translation "was
pprop "were "rules [was]

pprop "what "priority 0
pprop "what "rules [[!:in [what where] #] g10 [# !a:in [what where] #b] g11]
pprop "what "g10 [how]
pprop "what "g11 [[Tell me about :a :b.] [:a :b?]
                  [Do you want me to tell you :a :b?]
                  [Really.] [I see.] newkey]

pprop "where "priority 0
pprop "where "rules [how]

pprop "why "priority 0
pprop "why "rules [[# why don't I #stuff] g65
                   [# why can't you #stuff] g66 [#] g67]
pprop "why "g65 [[Do you believe I don't :stuff?]
                 [Perhaps I will :stuff in good time.]
                 [Should you :stuff yourself?] [You want me to :stuff?] how]
pprop "why "g66 [[Do you think you should be able to :stuff?]
                 [Do you want to be able to :stuff?]
                 [Do you believe this will help you to :stuff?]
                 [Have you any idea why you can't :stuff?] how]
pprop "why "g67 [[Why indeed?] [Why "why"?] [Why not?] how newkey]

pprop "wife "family "true

pprop "wish "belief "true

pprop "wont "translation "won't
```

```
pprop "xxyyzz "priority 0
pprop "xxyyzz "rules [[#] g50]
pprop "xxyyzz "g50 [[You're being somewhat short with me.]
                    [You don't seem very talkative today.]
                    [Perhaps you'd rather talk about something else.]
                    [Are you using monosyllables for some reason?] newkey]

pprop "yes "priority 0
pprop "yes "rules [[yes] g51 [#] g52]
pprop "yes "g51 [xxyyzz [pre [x yes] yes]]
pprop "yes "g52 [[You seem quite positive.] [You are sure.] [I see.]
                 [I understand.] newkey]

pprop "you "priority 0
pprop "you "translation "I
pprop "you "rules [[# I remind you of #] g44 [# I are # you #] g45
                   [# I # are #stuff] g46 [# I #stuff you] g47
                   [# I &stuff] g48 [#] g49]
pprop "you "g44 [dit]
pprop "you "g45 [newkey]
pprop "you "g46 [[What makes you think I am :stuff?]
                 [Does it please you to believe I am :stuff?]
                 [Perhaps you would like to be :stuff.]
                 [Do you sometimes wish you were :stuff?]]
pprop "you "g47 [[Why do you think I :stuff you?]
                 [You like to think I :stuff you, don't you?]
                 [What makes you think I :stuff you?] [Really, I :stuff you?]
                 [Do you wish to believe I :stuff you?]
                 [Suppose I did :stuff you. what would that mean?]
                 [Does someone else believe I :stuff you?]]
pprop "you "g48 [[We were discussing you, not me.] [Oh, I :stuff?]
                 [Is this really relevant to your problem?] [Perhaps I do :stuff.]
                 [Are you glad to know I :stuff?] [Do you :stuff?]
                 [What are your feelings now?]]
pprop "you "g49 [newkey]

pprop "you're "priority 0
pprop "you're "translation "I'm
pprop "you're "rules [[# I'm #stuff] g30]
pprop "you're "g30 [[pre [I are :stuff] you]]

pprop "your "priority 0
pprop "your "translation "my
pprop "your "rules [[# my #stuff] g25]
pprop "your "g25 [[Why are you concerned over my :stuff?]
                  [What about your own :stuff?]
                  [Are you worried about someone else's :stuff?]
                  [Really, my :stuff?]]

pprop "yourself "translation "myself
```