Ask Dr. Ruby

In 1950, Alan Turing, one of the founders of modern computer science, published a paper titled “Computing Machinery and Intelligence.” Electronic computers were just then starting to be used outside of math and science, and were being adopted by businesses and other organizations. There was widespread interest in this new technology, and people began to wonder just what these machines were capable of. The topic of Turing’s paper was the nature of intelligence and whether a machine could ever be considered truly intelligent.

In his paper Turing proposed a “thought experiment” that would address this question. In this experiment, which is now known as the Turing Test, a person is asked to carry on a conversation using a computer terminal. The person is told there are two other parties in the conversation, one a real person and the other a computer. The test is whether the person can tell which sentences were generated by a computer. The reason the conversation takes place using a computer terminal is to make it a fair test: if the conversation used telephones a person might notice that the computer’s voice was unnatural sounding, and since it will be a long time (if ever) before robots look and act like humans a face-to-face conversation would not make a fair test. In Turing’s test, the person is told there are two rooms, with a computer in one room and another person in the other room, and the person is allowed to direct a question to either room. If, after carrying out a prolonged discussion, the person could not tell which room held the computer then we could say the computer passed the test and should be considered intelligent.

It is interesting to note that 60 years after Turing’s paper was published his test is being applied in a real way. If you use an instant messaging service or participate in on-line chats at any of the popular social networking web sites you might have seen a posting by a “chatbot,” a computer program that generates posts. Malicious chatbots generate spam messages that fill chat rooms with advertising, and some chatbots have reportedly tried to pose as real users to fool people into revealing credit card numbers and other financial information.

One of the first programs to attempt to converse in English was named ELIZA. It was written by Joseph Weizenbaum, a computer scientist at MIT, in 1966. ELIZA was a remarkable program, in part because it was able to generate realistic English sentences, but mainly because of how people reacted to it. Weizenbaum was amazed by how often people were willing to open up and converse at length with ELIZA, even when they knew they were getting responses from a computer and not another human being. In Computer Power and Human Reason, a book he published in 1976, Weizenbaum compared this “ELIZA effect” to
going to a play. We know the people on stage are actors, but we suspend our disbelief for a few hours and think of the characters as real people with real lives. When people were typing sentences into ELIZA they seemed perfectly happy to carry on a conversation even when they knew it was a computer on the other end.

13.1 Overview of ELIZA

Weizenbaum’s idea for ELIZA was to make a program that would respond to sentences by performing a few simple operations based on rules for matching patterns in the input. To test this idea, he developed rules that would have his system mimic a Rogerian psychologist, so the rules were designed to look for words or phrases a person might use when talking to a psychotherapist.

The simplest type of rule is to have the program recognize key words and reply with one of several predefined responses for that word. For example, if the input contains “mother” or “father” or other family relationships, the response is one of several generic sentences about families:

E: My father wouldn’t buy me a puppy.
Tell me more about your family.

The word “computer” is another key word, and ELIZA will respond to any sentence containing the word “computer” with something like “Do computers worry you?” or “Don’t you think computers can help people?”

More complicated rules extract a large chunk of the input and use it as part of the response. As an example of how this works, consider statements like “I am sad because my team lost the big game” or “I am sad I never got a puppy.” In general, sentences that start out with “I am sad...” continue with a description or a reason the speaker is sad. A rule based on this type of sentence can simply find the words following “sad” and use them as part of the response:

E: I am sad I never got a puppy.
Can you explain what made you sad you never got a puppy?

In this second example, the key word is “sad.” ELIZA took the words following “sad,” in this case “I never got a puppy,” made a simple transformation that changed “I” to “you”, and used the result with its predefined response of “Can you explain what made you...” This second type of rule, where ELIZA takes a piece of the input and transforms it in order to produce the output, is typical of most rules, so in general we refer to the rules used by ELIZA as transformation rules.

Of course not all sentences that start out “I am sad...” fit this general pattern, and a rule can easily construct a response that looks pretty strange:

E: I am sad we never had a chance to play fetch.
I am sorry to hear you are sad we never had a chance to play fetch.

In the context of this conversation the “we” in the input sentence meant “my puppy and I,” but the “we” in the response has a different meaning – it looks like the ELIZA thought the speaker wanted to play with the computer.
The idea that a computer can carry on a conversation by applying rules to input sentences is the central concept in ELIZA. In the sections below we will explore the details of what defines a rule and how to create new rules, but for now the main idea is to realize that ELIZA’s ability to participate in a conversation depends entirely on the set of rules it uses. With only a few rules of the first type, where ELIZA always responds with set phrases associated with key words, a person interacting with ELIZA would soon realize they are conversing with a machine. But with more complex rules that analyze several parts of the input it’s possible to have ELIZA generate responses that give more of an appearance that it “understands” the person typing the sentences.

The set of rules used by ELIZA to transform an input sentence is called a script. Weizenbaum chose the name Eliza for his program to emphasize the fact that the rules in a script determine how the program will respond. Just as Eliza Doolittle the character became more refined by learning more rules of language and etiquette, ELIZA the program should become better at conversing as more rules are added to its script.

Weizenbaum envisioned a situation where different scripts could be written for different applications; for example, one might make a script with rules based on key words using in cooking and baking to make a system that could carry on a conversation with a chef. The original script that Weizenbaum wrote, and the one that made the program famous, was named DOCTOR. The DOCTOR script that played the role of Rogerian psychotherapist contained rules based on words like “sad,” “depressed,” “dream,” “wish,” and “family.”

The project for this chapter is based on a simplified implementation of ELIZA that has been written in Ruby. The program, named eliza.rb, can be downloaded from the web site. The first part of the tutorial will explore the structure of rules, to see how it is possible to identify key words and transform strings according to rules. We will then see how to write a script, which is a document containing a full set of rules. A script named doctor.ez, based on the set of rules from Weizenbaum’s original DOCTOR script, is also available from the web site.

The file eliza.rb contains the definition of a Ruby module named Eliza. To call the methods defined in this module, put the word Eliza in front of the method name. The load method defined inside the module is used to read a script, so to call this method to read the “doctor” script you would type Eliza.load("doctor.ez"). To start a conversation you would call Eliza.run.

**Tutorial Project**

Before you start this project download the program named eliza.rb and the script named doctor.ez from the web site and save it in the directory where you work on your tutorial projects.

T1. Start IRB, and tell it to load the ELIZA program:
   ```
   >>> load "eliza.rb"
   => true
   ```

T2. To start a conversation with ELIZA call the method named Eliza.run. When ELIZA is running the prompt changes to E: to let you know ELIZA is waiting for you to type a sentence:
   ```
   >>> Eliza.run
   E:
   ```

T3. Since ELIZA does not have a script yet, it can only apply a default rule that does not change the input. Everything you type will simply be echoed right back:

© 2008 John S. Conery. Please do not distribute without permission.
Figure 13.1: ELIZA is a Ruby program, stored in a file named eliza.rb. Like other programs used in this book, we can load it into our “Ruby workbench” and call methods defined in the program in order to run experiments. When eliza.rb is first loaded it does not have any transformation rules, but we can call Eliza.load, a method defined in eliza.rb, to read a script. After calling Eliza.load("doctor.ez") the rules in the file doctor.ez have been read into ELIZA and we can call Eliza.run to start a conversation. ELIZA will read a sentence, apply the rules in the script, and print the transformed sentence as its output.

E: Hello.
Hello
E: How are you?
How are you
E: Stop repeating what I say
stop repeating what i say
E: Hello.
Hello.
E: I need a puppy.
Why do you need a puppy?
E: I remember my cousin had a puppy.
Does thinking of your cousin had a puppy bring anything else to mind?
13.2 Using Regular Expressions to Define Sentence Patterns

T7. Try a few more sentences on your own. Don’t forget to type quit when you are ready to continue with the next problem in the tutorial.

T8. ELIZA uses key words to organize transformations. In the example describe earlier in this section, a sentence containing the word “mother” or “father” is transformed by a rule defined for family members to produce “Tell me more about your family.” To see a complete list of key words in the current script call Eliza.info:

   >> Eliza.info
   Script: doctor.ez
   28 rules
   Keys:
       alike, always, are, because, believe, brother, ...

T9. Start a new conversation by typing Eliza.run again, and make up some questions on your own. You can type anything you would like, but you are more likely to keep up the illusion of chatting with a psychologist if your statements include key words from the list printed by the info method.

13.2 Using Regular Expressions to Define Sentence Patterns

To implement a program like ELIZA in Ruby, one of the first decisions to make is how to look for patterns in sentences. The program needs to scan sentences like “My father wouldn’t buy me a puppy” to look for key words like “father,” and to scan sentences like “I’m sad I don’t have a dog” to break it into that parts before and after the key word “sad.”

The clear choice for how to do pattern matching in Ruby is to use regular expressions. It may seem like overkill when simply looking for key words – all we really need to do to search for key words is to call index or another method that does an exact match search – but as we will see regular expressions make it very easy to break sentences into parts and to reuse those parts when making more complex rules.

Recall from Chapter 10 that to find out if a string $s$ matches a pattern the simplest approach is to use the =~ operator. To see if a sentence contains a word all we need to do is make a regular expression using that word. For example, to see if the word “cow” appears in a sentence just see if the sentence matches /cow/:

   >> s = "I'm afraid of cows."
   => "I'm afraid of cows."
   >> s =~ /cow/  
   => 14

In this example the pattern is just a three-letter word, and the result of the =~ operator tells us the word was found starting at location 14 in $s$.

There is a problem with just turning a key word into a regular expression. We don’t want ELIZA to respond to “My cat is scowling at me” with something like “Tell me more about your farm,” which it might do with this sentence because it will find “cow” in the middle of the input string:

   >> "My cat is scowling at me." =~ /cow/  
   => 11
What we want is for ELIZA to look for “cow” as a complete word, but to not match the letters “cow” in the middle of “scowl.” A special construct called an anchor, introduced in Chapter 10, will solve this problem. Inside a regular expression, the two-letter sequence \b means “a word boundary must occur here.” In a regular expression like \bcow\b the c will match only if it is the first letter in a word, and the w will match only if it is the last letter in a word:

>> "That cow is scary." =~ /\bcow\b/  
=> 5
>> "My cat is scowling at me." =~ /\bcow\b/  
=> nil

Let’s return to the more complicated sentence pattern. Our goal is to write a pattern that will match sentences of the form “I’m sad...” and then to extract all the words following the key word “sad.” Regular expressions can help here, too. When a regular expression contains a pair of matching parentheses, Ruby saves all the characters from the input string in a variable that we can access later. For example, suppose the regular expression is /I’m .* cows/. This pattern matches any string that starts with the string I’m and ends with cows. The .* in the middle of the expression means “any sequence of characters.” Now suppose we enclose the .* in parentheses, so the new form of the expression is /I’m (.* ) cows/. This form of the expression matches exactly the same sort of strings as the original, but what it does is store all the parts of the matching string in a variable that we can access later. For reasons that will be explained below, the name of this variable is $1 (where the $ is part of the name). Here is an example:

>> s = "I’m afraid of cows."
>> s =~ /I’m (.* ) cows/  
=> 0
>> $1  
=> "afraid of"

The expression with the =~ operator succeeds, and returns 0, because the string s matches the expression beginning with the first character in s. The part of s that extends from the end of I’m to the start of cows is "afraid of". Since the .* is enclosed in parentheses, Ruby stores these letters in $1, where we can access them and use them to build a response.

In Ruby terminology, a part of a regular expression enclosed in parentheses is called a group. As an example of how a group can help with the “I’m sad” sentence pattern, recall what our goal is: we want to save all the characters following the word “sad,” so all we have to do is use a regular expression that puts (.* ) after the word “sad”:

>> "I’m sad my puppy was stepped on by a cow." =~ /I’m sad (.* )/  
=> 0
>> $1  
=> "my puppy was stepped on by a cow."

A regular expression can have any number of groups. When an expression has n groups, Ruby stores the matching parts of the input string in variables $1, $2, etc up to $n. Here is an example that matches the test sentence with a pattern that has three groups:

>> "I’m sad my puppy was stepped on by a cow." =~ /I’m (.* ) my (.* ) was (.* )/.  
=> 0
13.2 Using Regular Expressions to Define Sentence Patterns

Note that the parts of say that match the three groups are stored in $1, $2, and $3. This same regular expression can be used to match other strings:

```
>> "I'm relieved my final exam was canceled." =~ /I'm (.*) my (.*) was (.*)./
=> 0
>> $1
=> "relieved"
>> $2
=> "final exam"
>> $3
=> "canceled"
```

One particularly effective way to use groups in a regular expression is to give a list of alternative words we expect to find in an input sentence. For example, suppose we want to make a set of rules that will respond to fear of dogs. If a person types “I'm afraid of poodles” or “I'm afraid of pit bulls” the program can reply “Were you bitten by a dog?” but this response wouldn't work as well for “I'm afraid of cows.” We could make separate rules for each type of dog, but that would be awkward. What we want is a pattern that lists types of dogs, so we can use just one rule that is flexible enough to apply to many different but related types of input. The way to do this with a regular expression is to enclose the alternative names in parentheses, and separate them with a vertical bar character:

```
>> r = /I'm afraid of (pit bulls|poodles|terriers)/
=> /I'm afraid of (pit bulls|poodles|terriers)/
```

The expression above defined a regular expression and saved it in a variable named $r$ so we can use it more than once:

```
>> "I'm afraid of poodles" =~ $r
=> 0
>> $1
=> "poodles"
>> "I'm afraid of cows" =~ $r
=> nil
```

Our new expression matched the first sentence above because the word following “afraid of” was one of the alternatives listed between parentheses in the regular expression. Note also that since the set of alternatives is enclosed in parentheses the word from the input sentence that made the match is saved in $1.$

So now we know how to write regular expressions that describe general patterns for input sentences, and we know how to use those regular expressions to have Ruby break sentences into pieces and how to access those pieces. But there are still a few details left to attend to before we can put the pieces back together in order to create realistic looking responses. We will take a look at these details in the next section.
Tutorial Project

T10. Experiment with simple regular expressions that see if a word can be found in a sentence:

```ruby
>> "I like to read mysteries" =~ /like/
=> 2
>> "I like to read mysteries" =~ /read/
=> 10
```

Do you understand what the numbers printed by Ruby mean?

T11. To see why we have to put word break anchors around words type these two examples:

```ruby
>> "I dread having to take that test" =~ /read/
=> 3
```

T12. In that previous example Ruby found the letters “read” in the first sentence – they are part of the word “dread.” If the goal is to find the “read” as a complete word and not just those four letters anywhere in the input we have to put \b before and after the word:

```ruby
>> "I dread having to take that test" =~ /\bread\b/ 
=> nil
```

Make sure you understand that example: the regular expression /\bread\b/ means “the letters 'read' surrounded by word breaks.”

T13. Type these statements to experiment with the use of groups to capture the parts of the input string:

```ruby
>> "I like to read mysteries" =~ /\bread (.*)/ 
=> 10
>> $1
=> "mysteries"
>> "I dread having to take that test" =~ /\bread (.*)/ 
=> nil
```

T14. This expression matches a sentence with a pattern that has several groups:

T15. Save a regular expression that will match alternative words in a variable named r:

```ruby
>> r = /\b(read|write)\b/
=> /\b(read|write)\b/
```

T16. You can verify that r is a regular expression by asking Ruby to tell you which class r belongs to:

```ruby
>> r.class
=> Regexp
```

T17. Use the regular expression to match some test sentences:

```ruby
>> "I like to write poems" =~ r 
=> 10
>> $1 
=> "write"
>> "I dread having to take that exam" =~ r 
=> nil
>> "I read it all in one sitting" =~ r 
=> 2
>> $1 
=> "read"
```
13.3 Reassembly Strings

One way to understand how ELIZA creates responses is to think in terms of templates. The strings that ELIZA prints are defined in advance, with placeholders that need to be filled in with pieces from an input sentence. After using a regular expression to break an input into key words and the surrounding pieces ELIZA puts the pieces into placeholders in a template. When a regular expression has groups, the parts of the input that match the groups are inserted into the placeholders in the template.

As an example, suppose we want to make a rule that responds to sentences of the form “I'm glad my ... was ...” One response might be “Tell me more about your ...” where the dots in the response should be replaced by the word between “my” and “was” in the input. If someone types “I'm glad my CD player was working” the response based on this template would be “Tell me more about your CD player.”

Filling in the placeholders in a template is easy: all we have to do is use expression interpolation to insert the value of a variable into a string. Recall that if x is a variable, Ruby will replace $x with the value of x. Here is an example, using the variables produced by a regular expression:

```ruby
=> "Your phone is ringing." =~ /Your (.*) is (.*)/
=> 0
=> $1
=> "phone"
=> puts "Why are you interested in my #{$1}?"
Why are you interested in my phone?
```

The pattern matching operator =~ compared the string to the pattern, and as it did the letters "phone" were saved in $1. While evaluating the call to puts Ruby created a string to print, and when it made this string it inserted the current value of $1 in place of #{$1} to produce the result shown above.

Unfortunately this simple strategy doesn’t always work. Suppose we want the “doctor” to respond to statements of the form “I am ...” with “Are you really ...?” The regular expression is easy, but the straightforward method shown above for reassembling the parts doesn’t always give the right result:

```ruby
=> "I am crazy" =~ /I am (.*)/
=> 0
=> puts "Are you really #{$1}?"
Are you really crazy?

=> "I am out of my mind" =~ /I am (.*)/
=> 0
=> puts "Are you really #{$1}?"
Are you really out of my mind?
```

The problem in that second example is that Ruby stored “out of my mind” in $1 and then simply echoed this string as part of the response. A human doctor would have changed “my” to “your” so the response would be “Are you really out of your mind?”

In order to handle situations like this, ELIZA uses a strategy it calls postprocessing. After breaking the input into pieces with a regular expression, but before reassembling the pieces
with a template, ELIZA does an additional pattern matching operation on each of the variables $1$, $2$, etc. This second pattern matching step does single-word replacements. Every “I” is replaced with “you”, every “my” by “your”, and so on. The result isn’t perfect, but with a large enough set of replacement strings ELIZA can maintain the illusion.

The file eliza.rb has the definition of a class named Pattern that implements this idea of using a regular expression to match a sentence and then building a response using a template. The simplest way to make a Pattern object is to call Pattern.new and pass it a regular expression and a reassembly string. For example, this statement creates an object that represents the sentence pattern that transforms any sentence containing the word “father” into the canned response “Tell me more about your family”:

```ruby
p = Pattern.new(/father/, "Tell me more about your family")
```

The result printed by Ruby is just its way of summarizing the information stored in the Pattern object. To apply the pattern to a string, call its apply method:

```ruby
p.apply("My father didn’t laugh at my jokes")
```

If a pattern does not match a sentence the apply method will return nil:

```ruby
p.apply("My sister teased me.")
```

Here is another example of a Pattern object, showing how to use variables in the reassembly string:

```ruby
q = Pattern.new(/I am (.*)/, "Are you really $1?")
```

Two important details should be mentioned here:

- The Pattern class automatically wraps variable names inside braces. In the example above, the string passed to the constructor is _Are you really $1_ instead of _Are you really_amp_{$1}_.
- The class also adds word break anchors to regular expressions. If you pass _/father_ to the constructor, it attaches word breaks and turns it into _/bfather\b_/.

Both of these details are intended to make it easier to write (and read) patterns.

Up to this point we have been ignoring the problem of capitalization. Case is important when matching input sentences:

```ruby
"Father knows best." =~ /father/
```

But since key words used to defined patterns might appear at the front of a sentence, we want Pattern objects to be able to deal with capital letters. The apply method takes care of this automatically:
13.3 Reassembly Strings

```ruby
>> p = Pattern.new( /father/, "Tell me more about your family")
>> p.apply("Father knows best.")
=> "Tell me more about your family"
```

There is another thing ELIZA does to increase the realism of talking to another person. Instead of using just a single template for each sentence pattern, ELIZA keeps a list of templates and cycles through them one by one. For the example above, the first time someone enters a sentence of the form “I am ...” ELIZA responds with “Are you really ...?” but the next sentence that matches this pattern leads to a response of the form “Tell me why you think you are ...”

To make a pattern that has more than one reassembly string simply pass an array of strings to the constructor:

```ruby
>> q = Pattern.new(/I am (.*)/,
                   ['"Are you really $1?", "Do you think you are $1?"'])
=> /I am (.*)/ -> ['"Are you really $1?", "Do you think you are $1?"']
```

When apply is called it cycles through the list of responses:

```ruby
>> q.apply("I am crazy")
=> "Are you really crazy?"
>> q.apply("I am crazy")
=> "Do you think you are crazy?"
```

A Pattern object can also do postprocessing when it creates an output string. Recall from Chapter 2 that one of the other names for a Hash object is “associative map.” Hashes are often used to implement dictionaries that associate one word with another. To allow a Pattern object to postprocess a sentence it has constructed, pass a Hash as a parameter to the apply method:

```ruby
>> q.apply("I am out of my mind")
=> "Are you really out of my mind?"
>> q.apply("I am out of my mind", {"my" => "your"})
=> "Are you really out of your mind?"
```

When you pass a dictionary to the apply method, the Pattern makes an output sentence using one of its reassembly patterns, and then it scans the sentence. Each time it finds one of the Hash keys it replaces it with the corresponding value.

**Tutorial Project**

T18. Try doing a few regular expression matches and use the parts to create strings based on expression interpolation:

```ruby
>> r = /Time (.*)/;
=> /Time (.*)/
>> "Time flies" =~ r
=> 0
>> puts "I wrote 'time #$1'";
I wrote 'time flies'
=> nil
>> "Time is a magazine" =~ r
=> 0
```
puts "I wrote 'time #{\$1}'"
I wrote 'time is a magazine'
=> nil

**T19.** Make a simple pattern object that will respond to input sentences that include the word "time":

```ruby
p = Pattern.new(/time/,
  "Are you worried about time?"
)
=> /time/: ["Are you worried about time?"]

p.class
=> Pattern
```

**T20.** Use your new pattern on some test sentences:

```ruby
p.apply("I didn't have time to finish the exam")
=> "Are you worried about time?"

p.apply("Baseball is the national pastime")
=> nil

p.apply("My subscription to Time ran out")
=> "Are you worried about time?"
```

Make sure you understand why each of these examples produced the results they did.

**T21.** Next try a pattern that has groups in the regular expression and uses variable interpolation to produce the output string:

```ruby
p = Pattern.new(/I don't like (.*)/, "Why don't you like $1?"
)
=> /I don't like (.*)/: ["Why don't you like $1?"]

p.apply("I don't like cows")
=> "Why don't you like cows?"

p.apply("I don't like your attitude")
=> "Why don't you like your attitude?"
```

**T22.** Try a few sentences with this pattern, which matches sentences with any one of the words listed in parentheses in the regular expression:

```ruby
p = Pattern.new(/red|green|yellow|blue/, "Is $1 your favorite color?"
)
=> /red|green|yellow|blue/: ["Is $1 your favorite color?"]

p.apply("I wore a green shirt")
=> "Is green your favorite color?"

p.apply("She was feeling blue")
=> "Is blue your favorite color?"
```

**T23.** Make a pattern that has two response strings, and verify that the `apply` method alternates between the two responses:

```ruby
p = Pattern.new(/books|comics|web sites/,
  "Do you often read $1?", "Why do you mention $1"
)
=> /books|comics|web sites/: ["Do you often read $1?", "Why do you mention $1"

p.apply("The comics were funny today")
=> "Do you often read comics?"

p.apply("The comics were funny today")
=> "Why do you mention comics"

p.apply("The comics were funny today")
=> "Do you often read comics?"
```

### 13.4 Scripts

A script for **ELIZA** is a collection of rules that can be applied to sentences typed by the user. The simplest way to make a script would be to make a list of Pattern objects, and then the
13.4 Scripts

Figure 13.2: A script is a Hash object that associates words that might be found in sentences with rules that transform those sentences. Each rule is represented by a Rule object (shown here enclosed in dotted lines). Each rule is basically just an array of Pattern objects. In this example, if ELIZA sees the word “remember” in an input sentence, it first tries to see if the sentence matches the regular expression /I remember (.*)/. If so, it uses one of the reassembly strings in that pattern; if not, it tries the next pattern. Note that in this example the pattern for “dream” will always succeed; ELIZA simply responds to sentences containing the word “dream” by printing one of two canned replies.

program would just scan the list and apply the first pattern that has a regular expression that matches the input.

In order to more easily manage scripts that might grow to include hundreds of different rules ELIZA uses a more sophisticated technique for organizing scripts. When a user enters a sentence, ELIZA breaks it into individual words, and then it applies only the rules that are defined for those words. So, for example, if a person types “I’m afraid of cows” ELIZA would apply rules that have the words “I,” “afraid,” and “cow,” in their regular expressions, but it would not even try to apply rules with keywords like “father” or “computer.”

The class used to represent a rule in eliza.rb is named Rule. A Rule object is defined by a key word and a list of one or more Pattern objects that ELIZA will try to apply whenever the key is found in an input sentence. The script in eliza.rb is simply a Hash, a container that holds rules, where each Rule object is stored using its key word as its index (Figure 13.2).

To make a new Rule object call the constructor, passing it the key word:

```ruby
>> r = Rule.new("remember")
=> "remember" --> []
```

The output from IRB shows that r is a Rule object with an empty list of patterns. Next we can make some patterns with expressions that contain the word “remember:”

```ruby
>> p1 = Pattern.new(/I remember (.*)/)
=> /I remember (.*)/: []
```

We have an object p1 that is a new pattern, but it does not have any reassembly strings. We can add them with a method named addText:
Let's make a second pattern and add some reassembly strings to it:

```ruby
pt = Pattern.new(/do you remember (.*)/)
pt.addText("Did you think I would forget $1?"
= ["Did you think I would forget $1?"
pt.addText("Why do you think I should recall $1 now?"
= ["Did you think I would forget $1?", "Why do you think I should recall $1 now?"
```

Now we can add these two patterns to the new Rule object to make a rule that looks like the one in Figure 13.2:

```ruby
r.addPattern(pt)
```

Now if we ask Ruby to print the current value of `r` we'll see that it is a Rule object that associates the word “remember” with the two patterns that transform sentences that contain this word:

```ruby
r
= "remember" --> [
/I remember (.*)/
    "Do you often think of $1?"
    "Does thinking of $1 bring anything else to mind?"
/do you remember (.*)/
    "Did you think I would forget $1?"
    "Why do you think I should recall $1 now?"
]
```

The Rule class also has an `apply` method. When we call `apply` for a Rule object, the program tries to match the input sentence against each of the patterns by calling the pattern's own `apply` method. If none of the patterns match the sentence the method returns nil. The `apply` method for rules also takes an option parameter that is a list of postprocessing substitutions. Here are some examples:

```ruby
r.apply("I remember that movie")
= "Do you often think of that movie?"
```

Note that the last example returned nil because the string "I don’t remember" does not match either of the regular expressions /I remember/ or /do you remember/.
13.5 Script Files

Tutorial Project

If you have not already done so go back through the examples in this section. Enter the expressions that define a Rule object, add patterns to it, and then apply it to some sample sentences.

13.5 Script Files

Since a script may contain dozens, if not hundreds, of different rules, it would be a major inconvenience to have to create Pattern and Rule objects using IRB every time we wanted to experiment with ELIZA. As we saw in the first section, it is possible to put rules in a script file, and then load the script by calling Eliza.load.

Script files have a very simple syntax:

- A line can be a directive, which tells ELIZA something about the script, or it is part of a rule.
- Directives are words that start with a colon. For example, the line
  
  :post "my" "your"

  tells ELIZA that the line contains a pair of words to add to the collection of substitutions done during postprocessing.

- A transformation rule starts with a line that contains a single word. This word becomes the key for a new Rule object.

- Lines following the start of a rule can contain regular expressions or strings. A line containing a regular expression defines the start of a new Pattern object, and this object is added to the current rule. Lines containing strings are added to the current pattern as reassembly strings.

- The file can also contain comments, which are lines that start with a # character.

A portion of the “doctor script,” from the file doctor.ez, is shown in Figure 13.3.

13.6 Projects

- cooking (Weizenbaum’s idea)
- computer help desk – zero in on final answer (needs memory?)
- argument clinic

13.7 How Smart is Eliza?
# A portion of the "Doctor" script for ELIZA in Ruby
# (eliza.rb)
:start "How do you do. Please tell me your problem."
:stop "Goodbye."
:post me "you"
:post myself "yourself"
:post my "your"

was
/was i (.*)/
  "What if you were $1?"
  "Do you think you were $1?"
  "Were you $1?"
/i was (.*)/
  "Were you really?"
  "Why do you tell me you were $1 now?"

:alias $family mother father sister brother
my 2
/my .*($family) (\.*)/
  "Tell me more about your family."
  "Who else in your family $2?"

Figure 13.3: An excerpt from the “doctor” script. Lines beginning with '#' are comments, lines beginning with a word that starts with a colon are directives, and the remaining lines are all part of rule specifications.