Project 2: Eliza
Due: 7:00 PM, Nov 3, 2017

Contents

1 Introduction 1

2 Regular Expressions (Regexp) 1
   2.1 Any * Details ....................................................... 3
   2.2 Practice .............................................................. 5

3 How ELIZA Works 6
   3.1 An Example ........................................................... 7
   3.2 More Practice ........................................................ 7

4 Data Definition 8

5 Assignment 9
   5.1 Helper Procedures .................................................. 11
   5.2 Still More Practice .................................................. 11

6 Handing In 12
   6.1 Design Check ......................................................... 12
   6.2 Final Handin ........................................................... 13
   6.3 Grading ................................................................. 13

7 Interaction 14

8 Extra Features 14

1 Introduction

In 1966, Joseph Weizenbaum developed a program called “ELIZA”. It was intended to be a parody of a Rogerian psychotherapist, whose technique is to reflect patients' sentences back at them. When ELIZA was run, it requested an input from a human user, who would type English sentences into the terminal. ELIZA then responded to this input in a seemingly intelligent manner. A typical session with ELIZA might go as follows:

eliza> Hello, how are you?
human> I feel really tired today.
eliza> Why are you feeling really tired today?
human> My family is killing me.
eliza> Your family is killing you?
human> They make me work too hard.
eliza> Can you be more specific?
human> My siblings.
eliza> Tell me about your siblings.
human> My siblings keep telling me to put family first.
eliza> Do you think that is because of your childhood?

ELIZA responds as if it understands what the user is saying. But when it runs, the program is simply looking for certain patterns in the user's input. For example, in the conversation above, ELIZA observes that the first sentence input by the human begins with “I feel...” Upon seeing such an input, ELIZA is programmed to build a response that starts with “Why are you feeling...”. The rest of her response is then “really tired today”, which is precisely how the human said they were feeling.

Weizenbaum’s ELIZA was more complex than this. But even with this simple version, it’s possible to produce a conversation that appears almost human, just by looking for patterns and using the relevant parts of the sentence. That is what you will be doing in this assignment.

2 Regular Expressions (Regexp)

In a language like English, a sentence or phrase can be described as a sequence of words and punctuation marks. In this project, we’re going to define a phrase to be a sequence of words, where a word is a sequence of characters or apostrophes containing no blanks, or a piece of punctuation like a comma or question mark or period. (We don’t call this a sentence because sometimes a user will type something like “Just wow”, which might not really be a proper sentence, but “phrase” is still a reasonable description of it.)

We’ll be working a lot with phrases, so make sure you understand the definitions above.

Take note of the following phrases:

- I love the beach
- I really hate the sand
- I want to pet the dog
- I the those

You can probably see a pattern between these phrases. They all start with the word “I”, followed by any number of words (0 or more), followed by the word “the”, followed by one single word. A phrase (or “sentence”, which we’ll use as a synonym for phrase) may have some structure (such as the examples above or the “I feel...” sentence in the ELIZA example in the introduction) that we’d like to detect or describe. To do so, we’re going to use regular expressions.
A regular expression, or regexp, is a concise way of describing sequences of words that have some particular structure or pattern. Since a regular expression is a way to represent a pattern in this project, we use the words “regexp” and “pattern” interchangeably.

The regular expressions you will use in ELIZA are sequences (represented as lists) of components where each component is one of three things:

- A literal (i.e., a specific word such as “I” or “the”, or “!”), which matches only itself
- “One”, often spelled ‘?’, which matches a single word
- “Any”, often spelled ‘*’, which matches any number of words, (0 or more)

“One” and “Any” are commonly called wild cards, because they don’t have to match to a specific word, and instead can match to any arbitrary word.

When we say that a regular expression matches a sentence or phrase, we mean that every word in the phrase corresponds to a word or a wild card in the regular expression, in the correct order.

In the phrases above, we could use the following regular expression:

- “I”, Any, “the”, One

which we would write as:

- “I”, *, “the”, ?

As you can see, each of the phrases above matches this pattern

<table>
<thead>
<tr>
<th>“I”</th>
<th>*</th>
<th>“the”</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>love</td>
<td>the</td>
<td>beach</td>
</tr>
<tr>
<td>I</td>
<td>really hate</td>
<td>the</td>
<td>sand</td>
</tr>
<tr>
<td>I</td>
<td>want to pet</td>
<td>the</td>
<td>dog</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>the</td>
<td>those</td>
</tr>
</tbody>
</table>

The following table shows some examples of regexps and some phrases that they match. Remember that ? (One) matches any one word, and * (Any) matches zero or more words.
<table>
<thead>
<tr>
<th>Regexp</th>
<th>Matches Regexp</th>
<th>Does Not Match Regexp</th>
</tr>
</thead>
<tbody>
<tr>
<td>“I ? your Dog”</td>
<td>“I like your Dog”</td>
<td>“I don’t like your Dog”</td>
</tr>
<tr>
<td></td>
<td>“I pet your Dog”</td>
<td>“I pet your Cat”</td>
</tr>
<tr>
<td>“I * your Dog”</td>
<td>“I like your Dog”</td>
<td>“I like my Dog”</td>
</tr>
<tr>
<td></td>
<td>“I really like your Dog”</td>
<td>“I really like your Cat”</td>
</tr>
<tr>
<td>“The Fox * ran ? ?”</td>
<td>“The Fox and Falco ran very far”</td>
<td>“The Fox and Falco ran away”</td>
</tr>
<tr>
<td></td>
<td>“The Fox and Falco ran by foot”</td>
<td></td>
</tr>
<tr>
<td>“The Fox * ran *”</td>
<td>“The Fox ran”</td>
<td>“The Dog ran”</td>
</tr>
<tr>
<td></td>
<td>“The Fox and Falco ran”</td>
<td>“The Dog and Cat ran away”</td>
</tr>
<tr>
<td></td>
<td>“The Fox and Falco ran to”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“The Fox and Falco ran to town”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“The Fox and Falco ran to his home…”</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Some examples of regular expressions and phrases they match.

2.1 Any * Details

The regexp “we are *” matches the following phrases (see Table 3):

- “we are” (here “*” matches zero words)
- “we are here” (here “*” matches one word: “here”)
- “we are what we are” (here “*” matches three words: “what we are”)

But the following phrases do not match this regexp:

- “here we are” (“here” does not correspond to any word in the regexp; any phrases that matches must start with the two words “we are”)
- “we think we are going” (even though * can match “going”, there is no wild card at the start of the regexp to match “we think” or between “we” and “are” to match “think we”)

What happens when * is located between two words, as in “we * are”? You still have simple matches and more complicated ones: phrases that match the regexp “we * are” include “we are” (* matches zero words) and “we the people are” (* matches two words—“the people”).
<table>
<thead>
<tr>
<th>Regexp</th>
<th>Matches</th>
<th>Does Not Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>“we are *”</td>
<td>“we are _”</td>
<td>“here are”</td>
</tr>
<tr>
<td></td>
<td>“we are here”</td>
<td>“we can”</td>
</tr>
<tr>
<td></td>
<td>“we are what we are”</td>
<td></td>
</tr>
</tbody>
</table>

| “we * are”    | “we are”                        | “surely we are”                         |
|               | “we the people are”             | “we the people in order to”             |
|               | “we are what we are”            |                                        |

Table 2: Matches for regular expressions with one *. The underlined phrase match the *.

Now, does “we are what we are” also match “we * are”?

In trying to match “we are what we are” against the regexp “we * are”, you might start by thinking to yourself “Okay, well, * can match any sequence of zero or more words, so let’s first try letting * represent zero words.” Then you might think, “The first word in the rule is ‘we’ and the first word in the phrase is ‘we’—looks good. The second word in both is ‘are’—still looking fine.” But then you’d think, “Uh oh! The phrase goes on, but the regexp ends. This phrase can’t match the regexp, because all of the words in the phrase can’t be matched in the regexp.” And you’d be partly right, but only because you were working under the assumption that * matched zero words. You would reach the same conclusion if you tried to match the regexp letting * represent one or two words. But if you let * stand for three words—“are what we”—then they do match.

<table>
<thead>
<tr>
<th>Regexp</th>
<th>“we * are”</th>
</tr>
</thead>
<tbody>
<tr>
<td>This way seems to match...</td>
<td>“we are what we are”</td>
</tr>
<tr>
<td>...But this way doesn’t...</td>
<td>“we _ are what we are”</td>
</tr>
<tr>
<td></td>
<td>(the last 3 words, “what we are” doesn’t match)</td>
</tr>
</tbody>
</table>

Table 3: Matches for regular expressions with one *. The underlined phrases match the pieces of the pattern, with the middle one matching the *.

So yes, “we are what we are” also matches “we * are” (* matches “are what we”). But the upshot of this discussion is really this: there can be various ways to compare a regexp and a phrase, some of which lead to a match and some of which do not. **If there’s any comparison that leads to a match, we say that the phrase matches the regexp** (or vice versa).

The regexp “we * are” has only one *. Things can get really hairy when there are multiple *s in a regexp (see Table 4):

- The phrase “you have got mail” matches the regexp “you have * *”, but there is more than one way to match “got” and “mail” to the *s. Both words could be covered by the first, both by the second, or “got” by the first and “mail” by the second. (But you cannot match “mail” to the first and “got” to the second!)

- The phrase “you have to have mail” matches the regexp “you * have *”, again in more than one way. You could match the first * with “have to” and the second * with “mail,” or the first * with nothing and the second * with “to have mail.”
<table>
<thead>
<tr>
<th>Regexp</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>“you have * *”</td>
<td>“you have got mail .”</td>
</tr>
<tr>
<td></td>
<td>“you have _ got mail”</td>
</tr>
<tr>
<td></td>
<td>“you have got mail”</td>
</tr>
<tr>
<td>“you * have *”</td>
<td>“you have to have mail”</td>
</tr>
<tr>
<td></td>
<td>“you _ have to have mail”</td>
</tr>
</tbody>
</table>

Table 4: Matches for regular expressions with two *s. The underlined phrases match the *s.

The point is that you have to remember that * can match zero or more words. So when matching regular expressions with *s to phrases you must be sure to check all possibilities (e.g., zero words, one word, and so on).

### 2.2 Practice

Before proceeding with the project, here are some practice problems you should do to make sure you really understand how * and ? matching work.

1. Determine whether the following phrases match the regexp “I * you”. For each phrase that matches, identify what portion of the phrase matches each component of the regexp (i.e., what matches “I”, what matches * and what matches “you”), by circling the three parts.
   - “I you”
   - “I love you”
   - “I love only you”
   - “I love to love you”

2. Determine whether the following phrases match the regexp “I * the ?”. For each phrase that matches, identify what portion of the phrase matches each component of the regexp (i.e., what matches “I”, what matches *, what matches “the”, and what matches ?), again by circling.
   - “I am the TA”
   - “I like the book”
   - “I do not like the book”
   - “I think the TA mimics the professor”

3. Some regular expression languages have an “AtLeastOne” wild card (often spelled ‘+’), which matches any sequence of literals with at least one element. For example, consider the regexp “I heart +”. The phrases “I heart NY” and “I heart my dog” match this pattern, but the phrase “I heart” does not.
   However, + is superfluous. Any rule which contains + can be rewritten without using +. Can you see how?
A note for the curious: regular expressions, which may have many more wildcards than just ‘*’ or ‘?’, are used all over computer science. They’ll show up if you take a Theory of Computation course. They’re used in many editors. They’re part of almost every ‘shell’, which you’ll learn about in CS33. They’re even used to describe architectural structures in some graphics programs for creating models of buildings, neighborhoods, or whole cities.

3 How ELIZA Works

Your implementation of ELIZA will involve two steps:

- Creating a list of rules, where each rule in the list will be a pair consisting of a regular expression and a corresponding template for building a response to any user input that matches that regular expression.

- Creating an eliza_respond procedure that consumes
  - A list of rules like the one you just created, and
  - A phrase, represented as a list of words. Recall that a word will be either (i) a sequence of letters, hyphens, and apostrophes, or (ii) a single punctuation mark like a comma, period, exclamation mark, or question mark.

Using the list of rules, your eliza_respond procedure will do two things:

1. If the user’s input matches with some regular expression in the list of rules, then the portions of the user’s input that match the wild cards in the regular expression will be extracted. Recall that the wild cards are Any (*) and One (?)

2. Second, a response to the user’s input will be constructed using the extracted portions of the user’s input and the response template that corresponds to the matched regular expression.

The response will be another phrase, again represented as a list of words. See below for an example and specifics of these steps in action.

If the user’s input does not match with any of the regular expressions in the list of rules, it’ll produce an error, so the rule-list you build will need to include a rule that matches anything at all, just as a fall-back case.

We haven’t yet said what a “response template” is; we’ll discuss that shortly.

We recognize that it’s a bit of a pain to type even a short sentence like ["I"; "feel"; "good"; "!"] as a list of words of the kind that eliza_respond requires. For that matter, it’s not particularly easy to read the sentence in this form. In order to smooth out this process, we’ve written two short programs that allow for conversion from word lists like ["I"; "feel"; "good"; "]!"] to ordinary strings like “I feel good!”

Here are the procedures you’ll have access to:

- to_phrase which converts ordinary strings into phrases, and

- from_phrase which converts phrases back to ordinary strings.
Using these, you can produce a program which both consumes and produces strings like so:

```ocaml
let eliza (input : string) (rules : rule list): string =
to_phrase(eliza_respond (from_phrase input) rules));
```

### 3.1 An Example

Say the user enters:

*I love CS17 because the professor rocks*

The user-input would be represented as

`["I"; "love"; "CS17"; "because"; "the"; "professor"; "rocks"]`

but we’ll almost never again write it that way, because it’s just too annoying to read, and you’ll be using our to/from procedures to do the conversion anyhow.

The first procedure you will call is `extract`. On the condition that the input matches a regular expression in your list of rules, `extract` then extracts from the user’s input the portions that match each of the wild cards in this regular expression in such a way that the whole matches the pattern. You’ll want to return an `option`—Some of the extraction if the input matches, and `None` if it doesn’t.

If, for example, there is a regexp in your list of rules that looks like “I love ? because *”, then the `extract` procedure should return `Some` of `(["CS17"], ["the"; "professor"; "rocks"]).

Given this data, and the relevant template, a response can be constructed. Such a template might look like this: “Is it really because 2 that you love 1 ?”. The numbers in this template act as placeholders; they indicate that a particular phrase will be inserted in a particular place when you construct ELIZA’s response. More specifically, you will substitute for “1” that portion of the user’s input that matched the first wild card, for “2” that portion of the user’s input that matched the second wild card, and so on.

Following these steps, ELIZA would respond to the above input as follows:

*Is it really because the professor rocks that you love CS17?*

(Again, the response really is a list of words rather than a single string, but we won’t write it that way.)

### 3.2 More Practice

The rules in the list can be arbitrarily complex. See if you can figure out what the following rules and inputs would produce:

1. Rule: “I don’t *” → “Why don’t you 1 ?”
   Input: “I don’t think the TAs are working hard enough”

2. Rule: “* hit *” → “2 was hit by 1”
   Input: “a ball hit the wall”

   Input: “Luke I am your father”
4. Rule: “* mother *” → “Tell me about your mother”
   Input: “my mother is a wonderful mother”

   Input: “The rain in Spain falls mainly on the plain”

   Input: “Buffalo buffalo buffalo Buffalo buffalo”

4 Data Definition

To keep the idea of a “word” as either a single word (maybe with hyphens or apostrophes, but no
blanks or other whitespace) or a single punctuation mark (again with no whitespace), we start with

```plaintext
type word = string (* a single word or punctuation mark, no whitespace *)
```

A regexp_element is represented by the following variant type:

```plaintext
type regexp_element = Lit of word | One | Any
```

For example, these are regexp_elements:

- Lit "example" (* represents a literal *)
- One (* represents a wild card *)
- Any (* represents a wild card *)

Similarly, a response_element, the building block for responses, is defined as follows:

```plaintext
type response_element = Text of word list | Place of int
```

For example, these are response_elements:

- Text ["Are" ; "you" ; "feeling"] (* represents text in response template *)
- Place 2 (* represents a placeholder in response template *)
- Text ["today" ; "?"] (* represents text in response template *)
- Place 1 (* represents a placeholder in response template *)

A regular expression is a list of regexp_elements, and a response template is a list of response_elements. We want you to work with the additional data types phrase, pattern, and response_template, purely for design and readability. While practically, these types are no different from a word list, regexp_element list or response_element list, respectively, creating these data types allow a reader to look at your code and easily understand the intention behind your procedures.

```plaintext
type phrase = word list
```

```plaintext
type pattern = regexp_element list
```
type response_template = response_element list

Note: We are using variant types to represent regexps. In particular, the wildcard ？ is represented by One in ELIZA and the wildcard * is represented by Any. That way, "?" can be used as a literal.
Recall that a rule is a pair that consists of a regular expression and a response template. Hence, a rule is defined as follows:

type rule = Rule of pattern * response_template

For example, these are rules:

Rule([Lit("I") ; Any ; Lit("my") ; Any],
    [Text(["Why" ; "do" ; "you"]); Place(1); Text(["your"]; Place(2);
    Text(["?"]))])

Rule([Lit("I") ; Any ; Lit("hate") ; Lit("you"],
    [Text(["Why" ; "do" ; "you"]); Place(1); Text(["hate" ; "me" ; "?")])

Rule([Any],
    [Text(["What's" ; "on" ; "your" ; "mind" ; "?"))))

You can find a file with a small list of sample rules at: /course/cs017/src/eliza/eliza_rules.ml.

5 Assignment

Your primary assignment is to write a procedure, eliza_respond that consumes a sequence of words and punctuation (what we’ve been calling a "phrase", but which is typically an English-language sentence!) and a list of rules and produces a response based on the rules.

We’ll also ask that you hand in a list of rules that can be used with your procedure.

To get started, download the eliza-starter.zip file and unzip it; TAs can help you do this for your own computer or on the department systems. The result of unzipping will be a folder containing all the files you need to hand in, with some of the material here to guide you (i.e., the first line of the README.txt file will say something like "replace this line with your and your partner's logins").

More specifically, your eliza_respond procedure should consume a phrase and a list of rules (in that order), and should produce a phrase. For example:

eliza_respond ["I" ; "like" ; "my" ; "cat"] eliza_rules
=> ["Why" ; "do" ; "you" ; "like" ; "your" ; "cat" ; "?"]

Start your post-design-check work by downloading the file eliza.ml located at /course/cs017/src/eliza/, which has template code for you to fill in, and contains our implementations of to_phrase and from_phrase.
This file will be made available once design checks are complete.

In total, your procedure, `eliza_respond` must do the following:

1. Take in the input `phrase` (the list of strings) and match it to one of the `rules` in the second input, the list of `rules` (which you will write in the `eliza_rules.ml` file you downloaded).
   - To help you do this step, consider the procedure
     `regexp_match : phrase → pattern → bool`, which determines whether an input matches a regexp.
     Examples:

     ```plaintext
     regexp_match ["cs" ; "17" ; "rocks"]
     [Lit("cs") ; Lit("17")]
     => false

     regexp_match ["cs" ; "17" ; "rocks"]
     [Lit("cs") ; Lit("17") ; One]
     => true

     regexp_match ["cs" ; "17"; "rocks" ; "my" ; "socks" ; "off"]
     [Lit("cs") ; Lit("17") ; Any]
     => true
     ```

   - If an input matches more than one rule, choose the first one. This lets you put specific rules at the beginning of your rule set and more general ones later, so you can generate more appropriate responses.
   - Remember that you will need to make your own rules in addition to the rules given by CS17 to exhaustively test your procedure, as required for your Design Check. You will want rules that match only literals, only `One`s, and only `Any`s, as well as many different combinations of the three.
   - Note that the example rules we provide in `eliza_rules.ml` use the regexp `Any` as a catch-all. That is, if no other regexp matches the user's input, it will match `Any`. This is an easy way to handle inputs that do not match any rule.
   - A good test case for your matching algorithm is this: try to match the input “I hate to hate you” with the regular expression “I * hate you”. Some matching strategies that appear to be working on most inputs will get this one wrong.
   - The list of rules that you are matching against should be located in your `eliza_rules.ml` file. Make sure that there are no type definitions in this file. To reiterate, you should write `#use eliza_rules.ml` at the bottom of your `eliza.ml` file, after all your type declarations and before your test cases.

2. extract the portions of the phrase using the wild cards in the pattern of the matched rule. See section 5.1 for more details on the `extract` procedure.
   - There are times when a user’s input will match a regexp in more than one way. Your version of ELIZA may choose any of them and still be correct. However, some matching strategies are easier to implement than others.

3. Using the extracted portions of the previous step and the `response_template` of the matched rule, return a response constructed by substituting the extracted literals into their positions in the template. See section 5.1 for more details on the `make_response` procedure.
5.1 Helper Procedures

Your `eliza_respond` procedure should rely on two helper procedures, `extract` and `make_response`:

- **extract**: `phrase → pattern → ((phrase list)option)`, which takes an input phrase and extracts the portions of the input that match each wild card, if the input matches the regular expression, and returns `None`, otherwise.

Examples:

```plaintext
extract ["cs" ; "17" ; "is" ; "really" ; "fun"]
    [Lit("cs") ; One ; Lit("is") ; One ; Lit("fun")]
=> Some ["17"] ; ["really"]
extract ["The" ; "right" ; "answer" ; "is" ; "right"]
    [Any ; Lit("right") ; Any]
=> Some ["The"] ; ["answer" ; "is" ; "right"]
or
=> Some ["The" ; "right" ; "answer" ; "is"] ; []
extract ["I" ; "am" ; "not" ; "me" ; "today"]
    [Lit("I") ; Any ; Lit("not") ; One]
=> None
```

**Hint**: You'll probably want to develop this procedure in stages. First, just handle regexps with only literals, then add support for `One`, and finally add support for `Any`.

- **make_response**: `(phrase list) → response_template → phrase`, which takes the output of `extract` and a response template as its two inputs, and returns a response constructed by substituting the extracted literals into their positions in the template.

Examples:

```plaintext
make_response ["Eric" ; "Alex"]
    [Text ["Why"; "was"] ; Place(2) ; Text ["hit"; "by"] ;
    Place(1) ; Text("?")]
=> ["Why"; "was" ; "Alex" ; "hit" ; "by" ; "Eric" ; "?"]
make_response ["cookies" ; "and" ; "cakes"]
    [Text ["What"; "do"; "you"; "like"; "about"] ; Place(1) ;
    Text("?")]
=> ["What"; "do"; "you"; "like"; "about"; "cookies" ; "and" ; "cakes" ; "?"]
```

Because we'll want to test not only your overall procedure, but these two helpers, you must use exactly these names and signatures for these two procedures.

5.2 Still More Practice

What would `extract` return in each of the following cases?
1. The regexp contains no Anys or Ones. Consider both when the input matches the regexp and when it doesn’t.

2. The regexp contains a total of $n$ wild cards. How many items will the output list contain? How many items will each list in the output list contain? Do these answers depend on the text? If so, how? If not, why not?

3. The input is $[]$, and the regexp is $[\text{Any}]$.

4. Can you think of any other regexps that $[]$ would match?

6 Handing In

6.1 Design Check

Design checks will be held on Oct 24-26, 2017. We will send out an email detailing how to sign up for design checks, so please check your inbox periodically, and sign up with your partner as soon as possible.

You should do the following to prepare for your design check:

1. Answer all the practice questions in Sections 2.2, 3.2, and 5.2 of this handout and bring them with you to the design check.

2. Come up with your own rule set consisting of at least eight rules (preferably creative and flexible ones) and bring them with you to the design check. Make sure that the rules you create thoroughly test regular expressions containing each of the two wild cards, as well as both of them together.

3. Complete the design recipe for all procedures, but not the implementation. This includes:
   
   (a) type/data definitions
   (b) type signatures
   (c) Input/Output contracts
   (d) templates—this means you should have all of the cases for a procedure written. In other words, all the different patterns that you’ll be matching on the left should be listed. Each case can return a default value for now, like None or the empty list.
   (e) test cases

4. Be prepared to walk the TA through your planned extract procedure.

5. Be prepared to describe to the TA how your make_response procedure will generate a response, given an extracted list and a response template.

The design check will be worth 25% of your grade.
6.2 Final Handin

Due: 7:00 PM, Nov 3, 2017. For the final handin, you are required to hand in three files: a ‘README.txt’ file, an ‘eliza.ml’ file, and a separate eliza_rules.ml file that contains your rules.

The ‘eliza.ml’ file must include eliza_respond and any helper procedures you wrote. All your code should be fully commented.

The eliza_rules.ml file should contain at least 8 exhaustive rules, as detailed in your design checks; these should be given a name, my_rules, so your file should begin

```ml
let my_rules = [ ... ];;
```

In the ‘README.txt’ file, you should provide:

- The logins of you and your partner on the very first line.
- instructions describing how a user would interact with your program (what would they input? what would they expect as output?)
- an overview of how all the pieces of your program fit together (when a user provides an input, what series of procedures are called in order to produce an output?)
- a short description of any possible bugs or problems with your program
- a list of the people with whom you collaborated
- a description of any extra features you chose to implement

To hand in your files, first navigate to the online handin website and select the eliza option.

6.3 Grading

The design check counts for 25% of your grade. Specifically,

- Answers to practice problems: 5 points
- Set of Eliza rules: 5 points
- Design recipe for all procedures: 5 points
- Walkthrough of extract algorithm: 5 points
- Walkthrough of make_response algorithm: 5 points

Functionality counts for 65% of your grade. Specifically,

- extract: 40 points
  - correctly determines when an input matches a pattern: 17 points
- extracts correctly for matching input: 23 points
  - make_response: 15 points
  - eliza_respond: 10 points

Partial functionality merits partial credit.

10% of the total is reserved for style errors. You should always:

- Follow the design recipe, including data definitions and examples of the data, type signatures, specifications, and test cases, for all procedures you write.
- Spend time trying to find elegant solutions to your problems. Clearer solutions usually result in shorter code that is easier to read and debug.
- Follow the CS 17 OCaml style guide.
- Add comments to any code that would otherwise be unclear.

You will not lose credit for style if a TA is ever in doubt about what a section of your code is doing or how it works.

7 Interaction

This part of the project is strictly for fun. You can ignore this section and still get full credit for ELIZA. But it’s pretty cool (and, did we mention, fun?), so we recommend you give it a try.

Navigate in your terminal to the directory containing your completed eliza.ml file.

Type in eliza to start an interactive ELIZA session.

The eliza command starts a repeating process which reads in your input (which you type directly, without any quote-marks and with splitting it up as a list of words — our code splits it into a list for you!), processes it using your eliza_respond, prints the resulting value, and repeats, until you explicitly tell it to stop. We provide you with this loop because it uses some features of OCaml that we don’t teach in CS 17.

8 Extra Features

If you finish your project early, but want to keep playing with ELIZA, there are lots of additional features you can implement. Note that these are strictly for fun; they do not count towards your grade, and they certainly do not make up for missing features in the actual assignment. Email the TA list for some ideas of extra features!

References

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs017/feedback