1 Introduction

The purpose of this assignment is to write a complete Scheme program from scratch. The program is called Eliza, and mimics the interactions between a patient and their psychiatrist.

Before starting this assignment, set your DrScheme language level to Standard (R5RS).

All your function definitions should be pure, i.e. they should not use any of Scheme’s imperative features such as set!. Also, never use iteration, always recursion.

Every function should be commented. At the very least, the comments should state what the function does, which arguments it takes, and what result it produces.

You may make use of any of the functions in the standard Scheme library.

You should make your functions as simple and elegant as possible by introducing appropriate helper functions.

You will be graded primarily on correctness, elegance, clarity, and documentation.

Functions which you have defined for one problem can be used to solve subsequent problems.

This assignment is graded out of 100. The Scheme assignments are worth a total of 10% of your final grade. This assignment is worth 4% of your final grade.

2 Input

To do this part of the assignment you need to know a little about how characters are handled in Scheme. A character is represented like this:

```
#\a
#\b
...
#\1
...
#\space
#\newline
```

You can read a character and peek at (but not consume) the next character from standard input using these two functions:
(peek-char)
(read-char)

You can compare characters using eq?, as usual:

> (eq? #\A #\a)
#f
> (eq? #\A #\A)
#t

You can convert between upper and lower case:

> (char-downcase #\A)
#\a
> (char-upcase #\a)
#\A

You can convert a list of characters to a string:

> (list->string '(#\A #\B #\C))
"ABC"

1. Write a function (read-line) which returns a line of text read from standard input as a list of characters: [6 points]

   > (read-line)
   hi there cutie!

   (#\h #\i #\space #\t #\h #\e #\r #\e #\space #\c #\u #\t #\i #\e #\!)

The last character read will be a newline (#\newline), and it should be discarded.

2. Write a function (strip L) which takes a list of characters as input (read by the function read-line above, for example) and [6 points]

   (a) removes all space (#\space) characters from the beginning of the list,
   (b) removes all space characters from the end of the list,
   (c) replaces any sequence of two or more space characters within the list with one space.

HINT: As always, it may make things a lot easier if you write some helper functions, and combine these into (strip L) itself.

Here are some examples:

> (strip '(#\h #\i #\space #\space))
(#\h #\i)
> (strip '(#\space #\space #\h #\i))
(#\h #\i)
> (strip '(#\space #\space #\h #\i #\space #\space #\space #\space #\space #\! #\space))
3. Write a function `(separate-specials L)` which takes a list of characters as input and adds a space character (`#\space`) before and after each "special character" (`,`, `,`, `.`, `!`, `?`). The reason for this function is that in a sentence such as `Hi there!` we want to treat the exclamation point by itself, not as part of the word `there`. [6 points]

Here is an example:

```lisp
> (separate-specials '(#\h #\i #\!))
(#\h #\i #\space #\! #\space)
```

4. Write a function `(tolower L)` which converts a list of characters to lower case: [6 points]

```lisp
> (tolower '(#\h #\i #\space #\T #\H #\e #\r #\e #\space #\!))
(#\h #\i #\space #\t #\h #\e #\r #\e #\space #\!)
```

You must implement this function using the built-in higher-order function `map`!

5. Write a function `(tokenize L)` which takes a list of characters as input, and returns a list of lists of characters: [6 points]

```lisp
> (tokenize '(#\h #\i #\space #\t #\h #\e #\r #\e #\space #\!))
((#\h #\i) (#\t #\h #\e #\r #\e) (#\space #\!))
```

`(tokenize L)` assumes that `L` has no initial or final space characters, and that there are not multiple spaces within the list. The output yields a separate sub-list for every "word" separated by a space.

6. Write a function `(char-list->symbol-list L)` which takes a list of lists of characters (such as was produced by `(tokenize L)` above) and returns a list of symbols: [6 points]

```lisp
> (char-list->symbol-list '((#\h #\i) (#\t #\h #\e #\r #\e) (#\space #\!)))
(hi there !)
```

Use the built-in function `string->symbol` to construct a symbol out of a string:

```lisp
> (string->symbol "hello")
hello
```
7. Using the functions `char-list->symbol-list`, `tokenize`, `strip`, `separate-specials`, `tolower`, and `read-line` defined above, you can now write the function `(read-sentence)` which reads a sentence from standard input and converts it to a list of symbols: [6 points]

   > (read-sentence)
   Hi there my name is Joel!

   (hi there my name is joel !)
   > (read-sentence)
   hi There  my name is JOEL...

   (hi there my name is joel |.| |.| |.)

3 Output

Scheme's built-in functions `(display)` and `(write)` don't give very pretty output of sentences. We therefore create our own output routines.

1. Write a function `(sentence->string L)` that converts a list of symbols (such as might have been produced by the function `(read-sentence)` above) and returns the corresponding string: [6 points]

   > (sentence->string '(hi there !))
   "hi there !"
   > (sentence->string (read-sentence))
   Hi There !

   "hi there !"

2. Write a function `(write-sentence S)` which prints out a list of symbols (such as might have been produced by the function `(read-sentence)` above): [6 points]

   > (write-sentence (read-sentence))
   HI thEre!
   hi there !

4 Patterns

Our program will rely on `pattern-matching` to answer questions from the user. The patterns are represented as a list of pairs of lists, like this:

```scheme
(define patterns
  '(
    (pattern-1 response-1)
    (pattern-2 response-2)
    ...
    (pattern-n response-n)
  )
)```
Patterns can contain the *wildcards* * and +. Both can match zero or more words. For example, the pattern

`( * mother * \. )`

matches any sentence that has the word mother in it and ends with a period (.). Similarly, the pattern

`( * likes to + her * \. )`

matches the following sentences

Lisa likes to pet her cat.
Lisa likes to pet her pretty black cat.
Lisa likes to blow-dry the fur of her pretty black cat.

The response-part of a (*pattern response*)-pair is also a sentence, that can optionally contain the wildcard +. This wildcard is replaced by whatever the corresponding + in the pattern matched. For example, the (*pattern response*)-pair

`(( * i remember + \. ) (do you often think about + ?))`

would match the sentence

i remember making applesauce.

and should generate the response

do you often think about making applesauce?

since the + in the pattern matched making applesauce.

Here is a definition of a list of patterns:

(define patterns
   `(
      (( * mother * \. ) (tell me more about your family \.))
      (( * i remember + \. ) (do you often think about + ?))
      ((i want + \. ) (what would you do if you got + ?))
      (( * likes to + her * \. ) (does she ever let her friends + her ?))
      (( * i + you * \. ) (perhaps in your fantasy we + each other \.))
      (( * \. ) (really ? tell me more !))
   )
)
1. Define a variable `patterns` (as above) containing your own set of patterns. You should have at least six. You can take inspiration from here: [http://www.stanford.edu/group/SHR/4-2/text/dialogues.html](http://www.stanford.edu/group/SHR/4-2/text/dialogues.html). To simplify the implementation we make the following restrictions: [6 points]

(a) Every sentence and every pattern must end with one of the punctuation characters ! . ?. Note that the period (.) has special meaning in Scheme and must be escaped (i.e., write \. in your patterns).

(b) The last pattern should be a “catch-all” pattern that matches everything.

(c) Use only lower-case characters.

(d) A pattern can only contain one +-wildcard, but multiple *-wildcards.

(e) Two wildcards must be separated by at least one word.

5 Matching

In this part we define the predicates that do the actual work of our Eliza program. They are essentially very simple pattern-matching functions, similar to regular-expression pattern matchers you may have seen elsewhere.

1. Define a function `(replace orig-sy repl-list L)` which replaces every occurrence of the symbol `orig-sy` in the list `L` with the elements from the list `repl-list`: [6 points]

   ```scheme
   > (replace '+ '(oh my) '(oh me + what a lot of funny things go by !))
   (oh me oh my what a lot of funny things go by !)
   > (replace '+ '(oh my) '(+ + +))
   (oh my oh my oh my)
   > (replace '+ '(oh my) '(hi there !))
   (hi there !)
   ```

2. Write a function `(matches? pattern sentence)` that returns `#t` if `sentence` matches `pattern`, and `#f` otherwise. The pattern can contain wildcards as defined above. Both the pattern and the sentence must end with a punctuation character. Here are some examples: [8 points]

   ```scheme
   > (matches? '(* mother \.) '(my little mother \.))
   #t
   > (matches? '(+ mother \.) '(my little mother \.))
   #t
   > (matches? '(+ mother * \.) '(my little mother is cute \.))
   #t
   > (matches? '(+ mother + to * \.) '(my little mother loves to cook applesauce \.))
   #t
   > (matches? '(+ mother \.) '(my little mother is cute \.))
   #f
   > (matches? '(* father \.) '(my little mother \.))
   #f
   >
   ```

3. Write a function `(match pattern sentence)` that is similar to the function `matches?` above, except[8 points]
(a) (match pattern sentence) should only be called when sentence is known to match pattern, i.e. when (matches? pattern sentence) returns #t. In other cases the output is undefined.
(b) (match pattern sentence) returns the list of symbols that matched the + wildcard, or '() if there was no + in the pattern.

Here are some examples:

> (match '(+ mother \.) '(my little mother \.))
(my little)
> (match '(* mother + to * \.) '(my little mother loves to cook applesauce \.))
(loves)
> (match '(* mother + \.) '(my little mother loves to cook applesauce \.))
(loves to cook applesauce)

6 Interaction

In this section we define the functions that interact with the user.

1. Write a function (eliza patterns sentence) which takes a list of patterns (such as was defined by the variable patterns above) and a sentence as argument, and produces a sentence (a list of words) as result. The patterns should be tried one a time until one is found that succeeds. [6 points]

Here are some examples:

> (eliza '(((* mother + \.) (she + ?)))
(my little)
> (eliza '(((* mother + \.) (she + ?)))
(my little mother loves to cook applesauce \.))
(loves)
> (eliza patterns 'you know, I remember my mother cooking applesauce \.))
(tell me more about your family |.|)

The last example uses the variable patterns defined above.

2. Write a function (interact patterns) that takes a pattern-list as argument, reads a sentence from the user, computes a response, converts this to a string, and prints the result. Make use of the functions read-sentence, eliza, and write-sentence. [6 points]

Here are some examples:

> (interact patterns)
i love you very much.
perhaps in your fantasy we love each other .
> (interact patterns)
you are mean.
really ? tell me more !

3. Write a function (main) that repeatedly invokes the interact-function above, to provide a fake interaction between a patient and their psychiatrist: [6 points]

> (main)
i miss my MOTHER.
tell me more about your family.
she likes to pet her cat.
does she ever let her friends pet her?
i hate you so much.
perhaps in your fantasy we hate each other.
go away.
really? tell me more!

To simplify the implementation you are allowed to make this an infinite loop that has to be interrupted by pressing DrScheme's Stop button.

7 Submission and Assessment

The deadline for this assignment is non, Wed Sep 22. You should submit the assignment (a text-file containing the function definitions) electronically using the Unix command "turnin cs372.3 <files>". This assignment is worth 4% of your final grade.

Don’t show your code to anyone, don’t read anyone else’s code, don’t discuss the details of your code with anyone. If you need help with the assignment see the instructor.