Brandeis University

Computer Science (1)21b (Spring Term, 2018)

Structure and Interpretation of Computer Programs

Problem Set 2: Computer Psychiatrist

Due Thursday, February 15

Reading Assignment: Chapter 2, Sections 2.1, 2.2.

1 Homework Exercises

Exercise 2.4: Representing pairs as procedures.

Exercise 2.22: Bad implementations of `square-list`.

Exercise 2.25: `car` and `cdr` finger-exercises.

Exercise 2.26: comparing `cons`, `list`, and `append`.

Exercise 2.27: `deep-reverse`.

Exercise 2.28: `fringe`.

2 Laboratory Assignment: Doctor

This assignment deals with a drastically simplified version of a famous program called `doctor` that was developed at MIT in the early 1960s by Professor Joseph Weizenbaum. The `doctor` program engages in a "conversation" with the computer user, in which it mimics a nondirective style of psychotherapy. In the following sample conversation, the user's responses are preceded by the prompt symbol `**`, and the program's responses are shown in upper case. In order to simplify the program, we omit most punctuation and represent the responses of both the user and the machine as Lisp lists.

```
==> (visit-doctor 'Bill)
(HELLO, BILL)
(WHAT SEEMS TO BE THE TROUBLE?)

**(everyone hates me)
(WHY DO YOU SAY EVERYONE HATES YOU)

**(whenever i enter a room no one will look me in the eye)
(MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS)

**(they laugh at me behind my back)
(YOU SEEM TO THINK THEY LAUGH AT YOU BEHIND YOUR BACK)


**(they get together and talk about how they hate me)
(MANY OF MY PATIENTS HAVE TOLD ME THE SAME THING)

**(even my mother hates me)
(YOU FEEL THAT EVEN YOUR MOTHER HATES YOU)

**(even when i was a baby she said i was ugly and stupid)
```

```
(PLEASE CONTINUE)

**(she hit me all the time)
(WHY DO YOU BELIEVE SHE HIT YOU ALL THE TIME)
```

Although the `doctor` program seems to understand and reply to the user's remarks, in fact the program has just two simple methods for generating a response. The first method is to take the user's reply, change some first-person words like "I," "me," "my," and "am" to the corresponding second-person words, and append the transformed response to a qualifying phrase such as "why do you say" or "you seem to think." The second method is to ignore what the user types and simply respond with some sort of hedge like "please continue" or "many people have the same sorts of feelings." The program chooses one of these two methods at random.

Every interactive program, including the Lisp interpreter itself, has a distinguished procedure called a *driver loop*. A driver loop repeatedly accepts input, determines how to process that input, and produces the output. The `visit-doctor` procedure first greets the user, then asks an initial question and starts the driver loop.

```
(define (visit-doctor name)
  (write-line (list 'hello, name))
  (write-line '(what seems to be the trouble?))
  (doctor-driver-loop name))
```

The driver loop prints a prompt and reads in the user's response. If the user says `(goodbye)`, then the program terminates. Otherwise, it generates a reply according to one of the two methods described above and prints it.

```
(define (doctor-driver-loop name)
  (newline)
  (write '**)
  (let ((user-response (read)))
    (cond ((equal? user-response '(goodbye))
           (write-line (list 'goodbye, name))
           (write-line '(see you next week)))
          (else (write-line (reply user-response))
                (doctor-driver-loop name)))))

(define (reply user-response)
  (cond ((fifty-fifty)
         (append (qualifier)
                 (change-person user-response)))
        (else (hedge))))
```

The predicate `fifty-fifty` used in `reply` is a procedure that returns true or false with equal probability.

```
(define (fifty-fifty)
  (= (random 2) 0))
```

Qualifiers and hedging remarks are generated by selecting items at random from appropriate lists:

```
(define (qualifier)
  (pick-random '((you seem to think)
                 (you feel that)
                 (why do you believe)
                 (why do you say))))

(define (hedge)
  (pick-random
   '((please go on)
```

```
    (many people have the same sorts of feelings)
    (many of my patients have told me the same thing)
    (please continue))))
```

The basis for the procedure that changes selected first person words to second person is the following `replace` procedure, which changes all occurrences of a given `pattern` in a list `lst` to a `replacement`:

```
(define (replace pattern replacement lst)
  (cond ((null? lst) '())
        ((equal? (car lst) pattern)
         (cons replacement
               (replace pattern replacement (cdr lst))))
        (else
          (cons (car lst)
                (replace pattern replacement (cdr lst))))))
```

This is used to define a procedure `many-replace`, which takes as inputs a list `lst` together with a list of `replacement-pairs` of the form

$$(([pat_1][rep_1])\ ([pat_2][rep_2])\ \ldots)$$

It replaces in `lst` all occurrences of pat_1 by rep_1, pat_2 by rep_2, and so on.

```
(define (many-replace replacement-pairs lst)
  (cond ((null? replacement-pairs) lst)
        (else
          (let ((pat-rep (car replacement-pairs)))
            (replace (car pat-rep)
                     (cadr pat-rep)
                     (many-replace (cdr replacement-pairs)
                                   lst))))))
```

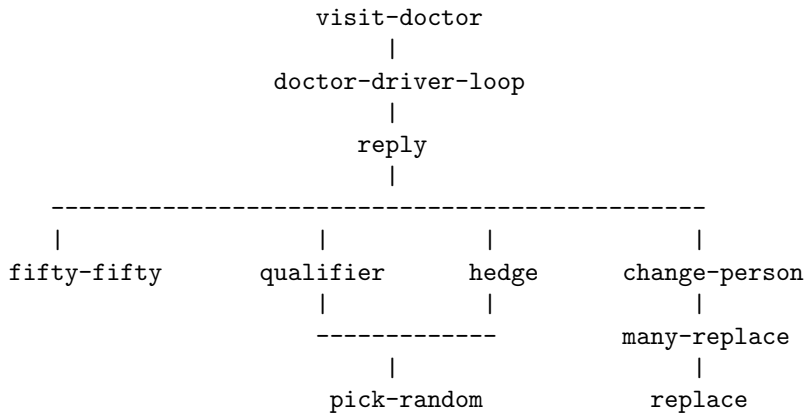
Note that `cadr` is a built-in function for `(lambda (x) (car (cdr x)))`. Changing the selected words is accomplished by an appropriate call to `many-replace`:

```
(define (change-person phrase)
  (many-replace '((i you) (me you) (am are) (my your))
                phrase))
```

The procedure `pick-random` used by `qualifier` and `hedge` picks an element at random from a given list:

```
(define (pick-random lst)
  (nth (1+ (random (length lst))) lst))
```

The following figure shows the pattern of procedure calls indicated in the text of the `doctor` program.



Procedure calls in the `doctor` *program.*

Problem 1. Edit the `qualifier` and `hedge` procedures to increase the doctor's repertoire of qualifying and hedging phrases. That is, make up and use some more of these phrases. Run through a brief session with the modified program.

Problem 2. What is the result of evaluating

```
(change-person '(you are not being very helpful to me))
```

We can improve the `doctor` program by having it not only change first person words to second person, but also second person to first. For instance, if the user types

```
(you are not being very helpful to me)
```

the program should respond with something like

```
(YOU FEEL THAT I AM NOT BEING VERY HELPFUL TO YOU)
```

Thus, "are" should be replaced by "am," "you" by "i," "your" by "my," and so on. (We will ignore the problem of having the program decide whether "you" should be replaced by "i" or by "me.")

- One idea for accomplishing this replacement is to simply add the pairs `(are am)`, `(you i)`, and `(your my)` to the list of pairs in the `change-person` procedure. Edit the procedure to do this. Now try evaluating

  ```
  (change-person '(you are not being very helpful to me))
  ```

  What does the modified procedure return? Does it matter whether you add the new pairs to the beginning or the end of the replacement list?

- In a few sentences, carefully describe the "bug" in the method of implementing replacement used above.

- Design a correct replacement method that will accomplish both kinds of replacement (first person by second person as well as second person by first person) and write a few sentences describing your strategy.

- Write, test, and debug procedures that implement your replacement strategy. Install these in the `doctor` program. Turn in, for this part, a listing of the procedures that you wrote, together with some sample responses that demonstrate that your method works.

Problem 3. Another improvement to the `doctor` program is to give it a third method of generating responses. If the doctor remembered everything the user said, then it could make remarks such as `(EARLIER YOU SAID THAT EVERYONE HATES YOU)`. Add this method to the program as follows.

- Modify the program so that the procedure `doctor-driver-loop` maintains a list of all user responses.[1]
- Modify the program so that `reply` occasionally replies by picking a previous user response at random, changing person in that response, and prefixing the modified response with "earlier you said that." If you want more control over how often the program uses each response method, you can use the following predicate, which returns true `n1` times out of every `n2`:

  ```
  (define (prob n1 n2)
    (< (random n2) n1))
  ```

  Turn in a listing of the procedures you wrote for this part.

Problem 4. The doctor currently sees only one patient, whose name is given in the call to `visit-doctor`. When that patient says `(goodbye)`, `visit-doctor` returns to the Scheme interpreter. Modify the program so that the doctor automatically sees a new patient after the old one goes away, and provide some way to tell the doctor when to stop. For example, `visit-doctor` might terminate after seeing a particular number of patients (supplied as an argument) or when it sees a patient with some special name (such as `suppertime`).

You may use the following procedure to find out each patient's name:

```
(define (ask-patient-name)
  (write-line '(next!))
  (write-line '(who are you?))
  (car (read)))
```

Now a session with the doctor might look like the following:

```
==> (visit-doctor)
(NEXT!)
(WHO ARE YOU?) (Groundhog)
(HELLO, GROUNDHOG)
(WHAT SEEMS TO BE THE TROUBLE?)

** (I am afraid of my shadow)
(WHY DO YOU SAY YOU ARE AFRAID OF YOUR SHADOW)
...

** (goodbye)
(GOODBYE, GROUNDHOG)
(SEE YOU NEXT WEEK)
(NEXT!)
(WHO ARE YOU?) (Peter Rabbit)
(HELLO, PETER)
(WHAT SEEMS TO BE THE TROUBLE?)

** (Mister MacGregor hates me)
...
```

[1] For people who have looked ahead: Do not use `set!` in order to implement this. It isn't necessary.

```
** (goodbye)
(GOODBYE, PETER)
(SEE YOU NEXT WEEK)
(NEXT!)
(WHO ARE YOU?) (suppertime)
(TIME TO GO HOME)
==>
```

Turn in a listing showing your modification.

Philosophical note: Do you think it is feasible to improve the program to the point where its responses are essentially indistinguishable from those of a real person? Some people have advocated using such programs in the treatment of psychiatric patients. Some patients who have participated in experiments claim to have been helped. Others object that such use of computers trivializes the human relationships that these programs caricature, and reinforces a tendency in modern society to debase human values.

If you are interested in the issues raised by the possibility of people interacting with computers as if they were "human," two excellent books to read are *Computer Power and Human Reason* (W.A. Freeman & Co., 1976) by Joseph Weizenbaum, and *The Second Self* (Simon & Schuster, 1984) by Sherry Turkle. My favorite introduction to the philosophy of cognitive science is *Minds, Brains, and Science* (Harvard University Press, 1986) by John Searle.

```
;; This is the code for ``Computer Psychiatrist'' (Doctor)

(define (visit-doctor name)
  (write-line (list 'hello, name))
  (write-line '(what seems to be the trouble?))
  (doctor-driver-loop name))

(define (doctor-driver-loop name)
  (newline)
  (write '**)
  (let ((user-response (read)))
    (cond ((equal? user-response '(goodbye))
           (write-line (list 'goodbye, name))
           (write-line '(see you next week)))
          (else (write-line (reply user-response))
                (doctor-driver-loop name)))))

(define (reply user-response)
  (cond ((fifty-fifty)
         (append (qualifier)
                 (change-person user-response)))
        (else (hedge))))

(define (fifty-fifty)
  (= (random 2) 0))

(define (qualifier)
  (pick-random '((you seem to think)
                 (you feel that)
                 (why do you believe)
                 (why do you say))))
```

```
(define (hedge)
  (pick-random '((please go on)
                 (many people have the same sorts of feelings)
                 (many of my patients have told me the same thing)
                 (please continue))))

(define (replace pattern replacement lst)
  (cond ((null? lst) '())
        ((equal? (car lst) pattern)
           (cons replacement
                 (replace pattern replacement (cdr lst))))
        (else (cons (car lst)
              (replace pattern replacement (cdr lst))))))

(define (many-replace replacement-pairs lst)
  (cond ((null? replacement-pairs) lst)
         (else (let ((pat-rep (car replacement-pairs)))
            (replace (car pat-rep)
                     (cadr pat-rep)
                     (many-replace (cdr replacement-pairs)
                     lst))))))

(define (change-person phrase)
  (many-replace '((i you) (me you) (am are) (my your))
                phrase))

(define (pick-random lst)
  (nth (1+ (random (length lst))) lst))

(define (nth n lst)
  (if (= n 1) (car lst) (nth (- n 1) (cdr lst))))

(define (atom? a) (not (pair? a)))

(define (prob n1 n2)
  (< (random n2) n1))

(define (ask-patient-name)
  (write-line '(next!))
  (write-line '(who are you?))
  (car (read)))
```