Segment 1

Scheme

These solutions are essentially the code in the practical script's Appendix, with the `???`s filled in.

Exercise 2 The top-level procedure needed to tack the word "the" on to the beginning of the two phrases. However, we can't just use `'the` since it is being joined to the *list* produced by the evaluation of `(noun-phrase)`. The joining operation is `append` which joins *lists*. So we need to specify a *list* containing the symbol `the`:

```
(define (sentence)
  (append (append '(the) (noun-phrase))
          (verb-phrase)
  )
)
```

Incidentally, we *could* have defined `(sentence)` slightly differently if we'd wanted to use `'the` instead of `'(the)`:

```
(define (sentence)
  (append (cons 'the (noun-phrase))              ;; Alternative definition
          (verb-phrase)
  )
)
```

□

Exercise 3 We want to pick an element between the first and last of `lst`. `list-ref` numbers the elements of a list from 0 to $l-1$, where l is the length of the list. `(random n)` gives a number between 0 and $n-1$, so the definition of `pick-random` just falls out:

```
(define (pick-random lst)
  (list-ref lst (random (length lst) ))
)
```

□

Exercise 4 With the above definitions, these are trivial!

```
(define (a-noun)
  (list (pick-random noun-list)))
```

```
(define (a-verb)
  (list (pick-random verb-list)))
```

□

Exercise 5 Since we've already got a procedure that evaluates one of two procedures at random (`either`), `noun-phrase` and `verb-phrase` are direct translations of their definitions.

```
(define (noun-phrase)
  (either a-noun an-adjectival-phrase)
)

(define (verb-phrase)
  (either a-verb an-adverbal-phrase)
)
```

The use of the two 'helper', or subsidiary, procedures `an-adjectival-phrase`, and `an-adverbal-phrase` simplify the `noun-phrase` and `verb-phrase` definitions — here we're using *abstraction* yet again to clarify the design. Notice how the recursion that arises from the definition of *noun phrase* appears in the program — `noun-phrase` uses `an-adjectival-phrase`, and *vice versa*. This is called *mutual recursion*.

```
(define (an-adjectival-phrase)
  (cons (pick-random adjective-list) (noun-phrase))
)

(define (an-adverbal-phrase)
  (append (a-verb) (list (pick-random adverb-list)))
)
```

The only difficulties you might have had with designing these procedures are likely to have been with when and whether to use `list`. Check that you understand how and why this has been used above.
□

1.1 Caring

I won't repeat the code that was given in the practical, so I assume that you're reading this in conjunction with the practical script.

Exercise 10 This is just the *hors d'oeuvres* ... there's really not much to do here, but I've done it all the same!

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

```
                (why do you say)
                (do you really feel that)
                (do you believe other people would say that)
                (how long have you felt that)
              )))

(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)
       (that's interesting)
       (Hmmm)
       (Uh-huh)
       (that's a really boring thought)
     )))
```

□

Exercise 11 The modification to `change-person` is simple. However, as you will have seen, there are problems when you try to use it.

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

□

Exercise 12 So, why didn't it work? There is a 'bug', but neither of the two procedures (`replace`, and `many-replace`) are actually *wrong*. It's just that together they don't do what we want!

The problem is that in the given arrangement, we are applying *each replacement* in turn to *all the elements* of the list, when in fact we need to apply *all replacements* to *each element* of the list in turn.

□

Exercise 13 There may be an elegant (and small) transformation from the given routines, but I'm not sure what it is ... hence the two redefinitions which are significantly different in structure from the originals.

```
(define (replace replacements x)          ;; replacements are (pat rep) pairs
  (cond ((null? replacements) x)
        ((equal? (car (car replacements)) x) (cadr (car replacements)))
        ( else (replace (cdr replacements) x))
  )
)

(define (many-replace replacements lst)
  (if (null? lst) nil
                (cons (replace replacements (car lst))
                      (many-replace replacements (cdr lst))
                )
 )
)
```

□

Exercise 14 We need to remember 'state' from loop to loop. The technique is to use extra parameters in a similar way to making recursive processes into iterative ones. We need to change `reply` so that it takes the list of previous responses as a parameter in order that second part of the exercise can be accomplished.

```
(define (doctor-driver-loop name response-list)
  (begin
    (newline)
    (display '**)
    (let ((user-response (read)))
      (cond ((equal? user-response '(goodbye))
                 (begin
                    (print (list 'goodbye name))
                    (print '(see you next week)))
            )
            (else (begin
                    (print (reply user-response response-list))
                    (doctor-driver-loop name (cons user-response response-list))
                  )
            )
      )
    )
  )
)
```

And we therefore have to modify `visit-doctor` to reflect the new signature[1] of `doctor-driver-loop`:

```
(define (visit-doctor name)
  (begin
```

[1] That is, the number and type of parameters.

```
    (print (list 'hello name))
    (print '(what seems to be the trouble?))
    (doctor-driver-loop name '() )))
```

The modification to `reply` is to use the given `prob` procedure to choose between the (now *three*) ways of responding. However, we must change the definition of `reply` to include the 'history' of user responses. Care has to be taken to ensure that the "Earlier you said that" response isn't tried before the patient has made an earlier response (i.e. when history is non-empty):

```
(define (reply user-response history)
  (cond ((prob 1 2) (append (qualifier) (change-person user-response)))
        ((and (prob 1 5) (not (null? history)))
                    (append '(earlier you said) (pick-random history)))
        (else (hedge))))
```

□

Exercise 15 This only require a couple of mods to `visit-doctor` — firstly to call `ask-patient-name` (in a `let` since we require the value in several places), then to check for the termination condition,[2] and finally to make the `visit-doctor` procedure into a loop.

```
(define (visit-doctor)
  (let ( (name (ask-patient-name)) )
    (if (eq? name 'AlanSugar) '(time to go home)
                (begin
                  (print (list 'hello name))
                  (print '(what seems to be the trouble?))
                  (doctor-driver-loop name '() )
                  (visit-doctor)
                ))))

(define (ask-patient-name)
  (begin
        (print '(next!))
        (display '(who are you?)) (display " ")
        (car (read))))
```

□

[2] What an utterly appropriate turn of phrase!

Exercise 16

Since this is meant to be your own invention, I'll leave this up to you. Let me know if you come up with anything magnificent.

However, here's the start of a solution to the first suggestion — you might like to take it forward from here.

```
(define (reply user-response history)
  (let ( (key-list (key-words user-response)) )
    (cond ((and (prob 1 2) (not (null? key-list)))
                   (canned-response key-words)
          )
          ((prob 1 2) (append (qualifier) (change-person user-response)))
          ((and (prob 1 5) (not (null? history)))
                    (append '(earlier you said) (pick-random history)))
          (else (hedge))
    )
  )
)

(define (key-words word-list)
        '()                              ; stub ... do this yourself
)

(define (canned-response key-words)
        '()                              ; stub ... ditto
)

(define response-list
  '(
     ( (depressed suicide)
         ( (When you feel depressed, have a pizza at Caesars)
           (depression is a disease that can be treated)
         )
     )
     ( (mother father parents)
         ( (tell me more about your parents)
           (why do you feel that way about your parents?)
         )
     )
   )
)
```

□