Unit5 Model Code Description

The experiments for this unit are a little more complex than the previous ones. They use only one new ACT-R function for creating perceptual information in experiments, but they also use several functions for interacting with the model more directly through code. The fan experiment includes an alternate version of the model running function (it is commented out of the experiment and thus not used) that will be discussed. The alternate function runs the model through the task without using the visual interface for input or the motor module for output. Instead it puts the input into the slots of the goal chunk before running the model and reads the response from a slot of the goal chunk upon completion. This is done through the use of functions that access chunks and buffers outside of the model. These mechanisms can be used when the details of the visual/motor systems are not of interest in the task being modeled and an abstraction of the experiment is acceptable for the objectives of the modeling task.

Here is the experiment code for the fan model. This experiment also differs from most that have been presented so far in that it only presents the test portion of the task. The study portion has been encoded in the model explicitly and if you want to do it as a person you would need to first study the required items. For the model, the experiment will favor one of two productions during two iterations over the task (once favoring one and then the other) so that the results can be averaged using only 2 runs. This is another case of the experiment being simplified for the tutorial. Another way to accomplish this would be to allow those two productions to compete equally and thus they would each fire on average half the time. Then the model would be run for many iterations over the task and the results would be averaged to generate the predictions, but because there is no noise in the model and the only source of differences are foil trials this simplification is sufficient for demonstration purposes.

```
(defvar *person-location-data* '(1.11 1.17 1.22
                                 1.17 1.20 1.22
                                 1.15 1.23 1.36
                                 1.20 1.22 1.26
                                 1.25 1.36 1.29
                                 1.26 1.47 1.47))

(defvar *response*)
(defvar *response-time*)
(defvar *model-doing-task* t)

;;#|
(defun fan-sentence-model (person location target term)
  (let ((window (open-exp-window "Sentence Experiment"
                                 :visible nil
                                 :width 600
                                 :height 300))
        (x 25))

    (reset)

    (install-device window)

    (case term
      (person (pdisable retrieve-from-location))
      (location (pdisable retrieve-from-person)))
```

```
   (dolist (text (list "The" person "is" "in" "the" location))
     (add-text-to-exp-window :text text :x x :y 150 :width 75)
     (incf x 75))

    (setf *response* nil)
    (setf *response-time* nil)
    (setf *model-doing-task* t)

    (proc-display)

    (run 30)

    (if (null *response*)
        (list 30.0 nil)
      (list (/ *response-time* 1000.0)
            (or (and target (string-equal *response* "k"))
                (and (null target) (string-equal *response* "d")))))))


;;|#

#| This version of the fan-sentence-model function
   runs the model without using the visual interface.
   It is provided as an example of showing how one could
   bypass the interface when it isn't really necessary.
   The difference is explained in the experiment description
   text.
|#
#|
(defun fan-sentence-model (person location target term)

  (reset)

  (case term
      (person (pdisable retrieve-from-location))
      (location (pdisable retrieve-from-person)))

  (mod-chunk-fct 'goal (list 'arg1 person 'arg2 location 'state 'test))

  (setf *response-time* (run 30.0))

  (setf *response* (chunk-slot-value-fct (buffer-read 'goal) 'state))

  (list *response-time*
        (or (and target (string-equal *response* "k"))
            (and (null target) (string-equal *response* "d")))))

|#

(defun fan-sentence-human (person location target term)
  (let ((window (open-exp-window "Sentence Experiment" :width 600 :height 300))
        (x 25)
        (start-time))

   (dolist (text (list "The" person "is" "in" "the" location))
     (add-text-to-exp-window :text text :x x :y 150)
     (incf x 75))

    (setf *response* nil)
    (setf *response-time* nil)
    (setf *model-doing-task* nil)
    (setf start-time (get-time nil))

    (while (null *response*) ;; wait for a key to be pressed by a person
           (allow-event-manager window))
```

```
    (list (/ (- *response-time* start-time) 1000.0)
          (or (and target (string-equal *response* "k"))
              (and (null target) (string-equal *response* "d"))))))


(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response-time* (get-time *model-doing-task*))

  (setf *response* (string key)))


(defun do-person-location (term &optional (in-order *actr-enabled-p*))
  (let ((test-set '(("lawyer" "store" t)("captain" "cave" t)("hippie" "church" t)
                    ("debutante" "bank" t)("earl" "castle" t)("hippie" "bank" t)
                    ("fireman" "park" t)("captain" "park" t)("hippie" "park" t)
                    ("fireman" "store" nil)("captain" "store" nil)("giant" "store" nil)
                    ("fireman" "bank" nil)("captain" "bank" nil)("giant" "bank" nil)
                    ("lawyer" "park" nil)("earl" "park" nil)("giant" "park" nil)))
        (results nil))

    (dolist (sentence (if *model-doing-task* test-set (permute-list test-set)))
      (push (list sentence
                  (apply (if *model-doing-task* #'fan-sentence-model #'fan-sentence-human)
                         (append sentence (list term))))
            results))
    (mapcar #'second (sort results #'< :key #'(lambda (x) (position (car x) test-set))))))

(defun fan-experiment (&optional who)
  (if (eq who 'human)
      (setf *model-doing-task* nil)
    (setf *model-doing-task* t))

  (output-person-location (mapcar #'(lambda (x y)
                                      (list (/ (+ (car x) (car y)) 2.0)
                                            (and (cadr x) (cadr y))))
                            (do-person-location 'person)
                            (do-person-location 'location))))

(defun output-person-location (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *person-location-data*)
    (mean-deviation rts *person-location-data*)
    (format t "~%TARGETS:~%                          Person fan~%")
    (format t  "  Location      1             2             3~%")
    (format t "    fan")

    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* i 3)) data))))

    (format t "~%~%FOILS:")
    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* (+ i 3) 3)) data))))))
```

First it defines a global variable that holds the experimental data to which the model will be compared.

```
(defvar *person-location-data* '(1.11 1.17 1.22
                                 1.17 1.20 1.22
                                 1.15 1.23 1.36
                                 1.20 1.22 1.26
```

```
                                    1.25 1.36 1.29
                                    1.26 1.47 1.47))
```

Then it defines the global variables that will hold the response, response time, and an indication of whether it is a human or the model doing the task.

```
(defvar *response*)
(defvar *response-time*)
(defvar *model-doing-task* t)
```

The fan-sentence-model function takes 4 parameters. The first, person, is the string of the person to display in the sentence. The second, location, is the string of the location to display. The third, target, is to be either t or nil to indicate whether this is a target or a foil trial respectively, and the fourth, term, specifies which of the productions to favor in retrieving the study item and should be either the symbol person or location. The function returns a list of two items. The first is the response time of the key press in seconds or 30.0 if there was no response and the second item is t if the response was correct (k for a target and d for a foil) or nil if there was no response or the response was incorrect.

```
(defun fan-sentence-model (person location target term)
```

First open a window.

```
 (let ((window (open-exp-window "Sentence Experiment"
                                :visible nil
                                :width 600
                                :height 300))
```

Create a variable to hold the x coordinates of the items to present.

```
        (x 25))
```

Reset the model and tell it which window to interact with.

```
    (reset)
    (install-device window)
```

Disable the retrieve production that is not being used this trial with the pdisable command described below.

```
    (case term
      (person (pdisable retrieve-from-location))
      (location (pdisable retrieve-from-person)))
```

Iterate over the words of the sentence and display each one 75 pixels from the previous one.

```
   (dolist (text (list "The" person "is" "in" "the" location))
     (add-text-to-exp-window :text text :x x :y 150 :width 75)
     (incf x 75))
```

Clear the response variables and indicate it is the model doing the task.

```
(setf *response* nil)
(setf *response-time* nil)
(setf *model-doing-task* t)
```

Make the model process the display and run for up to 30 seconds.

```
(proc-display)

(run 30)
```

If there was no response return the list of 30.0 and **nil** otherwise return the list of the response time in seconds and whether the response was correct or incorrect.

```
(if (null *response*)
    (list 30.0 nil)
  (list *response-time*
        (or (and target (string-equal *response* "k"))
            (and (null target) (string-equal *response* "d")))))))
```

Next is a block of comments (the #| and |# are the start and end markers for a block of comments in Lisp) which contain a description of a function and its definition. As indicated in the description this function is a replacement for fan-sentence-model which was described above. The version in the comments operates without using a display for interacting.

There is a second set of productions in the model file that are also initially commented out which run with the replacement fan-sentence-model function. It does not use the visual or motor modules, but the predictions are identical. The model is probably easier to generate and will run faster without the overhead of generating a display, but does require extra work to estimate the times for reading the screen and making the response since they will not be automatically produced by the respective modules.

Depending on the objectives of the modeling sometimes the actions aren't really important and using a simplified representation or direct manipulation of the chunks is sufficient. Prior to ACT-R 5.0, most models operated in such a manner because the perceptual and motor modules were not an integrated component of the system.

```
#| This version of the fan-sentence-model function
   runs the model without using the visual interface.
   It is provided as an example of showing how one could
   bypass the interface when it isn't really necessary.
   The difference is explained in the experiment description
   text.
|#
#|
(defun fan-sentence-model (person location target term)
```

First, reset the model and disable the unwanted retrieval production.

```
  (reset)
```

```
  (case term
      (person (pdisable retrieve-from-location))
      (location (pdisable retrieve-from-person)))
```

Next, modify the chunk named goal placing the strings directly into the slots. The mod-chunk-fct function will be discussed in more detail below. The chunk named goal will be placed into the goal buffer when the model runs by the goal-focus command in the model definition.

```
  (mod-chunk-fct 'goal (list 'arg1 person 'arg2 location 'state 'test))
```

Set the response time variable to the value returned by running the model.

```
  (setf *response-time* (run 30.0))
```

Set the response to the value in the state slot of the chunk in the goal buffer. The chunk-slot-value-fct function and the buffer-read function will be described below.

```
  (setf *response* (chunk-slot-value-fct (buffer-read 'goal) 'state))
```

Return the list of response time and correctness as above.

```
  (list *response-time*
        (or (and target (string-equal *response* "k"))
            (and (null target) (string-equal *response* "d")))))
|#
```

The fan-sentence-human function operates like the fan-sentence-model function except that the term parameter is ignored and it waits for a person to press a key instead of the model.

```
(defun fan-sentence-person (person location target term)
  (let ((window (open-exp-window "Sentence Experiment" :width 600 :height 300))
        (x 25)
        (start-time))

   (dolist (text (list "The" person "is" "in" "the" location))
     (add-text-to-exp-window :text text :x x :y 150)
     (incf x 75))

    (setf *response* nil)
    (setf *response-time* nil)
    (setf *model-doing-task* nil)
    (setf start-time (get-time nil))

    (while (null *response*) ;; wait for a key to be pressed by a person
           (allow-event-manager window))

    (list (/ (- *response-time* start-time) 1000.0)
          (or (and target (string-equal *response* "k"))
              (and (null target) (string-equal *response* "d"))))))
```

The key handler for the window just records the key press and the time of that press in the global variables.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response-time* (get-time *model-doing-task*))
```

```
  (setf *response* (string key)))
```

The do-person-location function takes one required parameter, term, which specifies which production to favor for the model as specified for test-sentence-model. It iterates over all of the test sentences for the task collecting the data as returned by the fan-sentence-model or fan-sentence-human function. If the model is performing the task then the items are presented in the default order, but if a person is doing the task the order is randomized. It returns a list of responses in the same order as the results in the global data list.

```
(defun do-person-location (term)
(let ((test-set '(("lawyer" "store" t)("captain" "cave" t)
                  ("hippie" "church" t)("debutante" "bank" t)
                  ("earl" "castle" t)("hippie" "bank" t)
                  ("fireman" "park" t)("captain" "park" t)
                  ("hippie" "park" t) ("fireman" "store" nil)
                  ("captain" "store" nil)("giant" "store" nil)
                  ("fireman" "bank" nil)("captain" "bank" nil)
                  ("giant" "bank" nil)("lawyer" "park" nil)
                  ("earl" "park" nil)("giant" "park" nil)))
      (results nil))

    (dolist (sentence (if *model-doing-task* test-set (permute-list test-set)))
      (push (list sentence
                  (apply (if *model-doing-task* #'fan-sentence-model #'fan-sentence-human)
                         (append sentence (list term))))
            results))

    (mapcar #'second (sort results #'< :key #'(lambda (x)
                                                (position (car x) test-set))))))
```

The fan-experiment function takes one optional parameter which indicates whether a person or model is to do the task. If it is specified as the symbol human then it will display the screen and wait for a person to respond, otherwise it will run the model through the experiment. It calls the do-person-location function once to favor the person and once to favor the location. It returns a list of responses that are the average of the two response times for a given condition and **t** if both responses were correct.

```
(defun fan-experiment (&optional who)
  (if (eq who 'human)
      (setf *model-doing-task* nil)
    (setf *model-doing-task* t))

  (output-person-location (mapcar #'(lambda (x y)
                                      (list (/ (+ (car x) (car y)) 2.0)
                                            (and (cadr x) (cadr y))))
                            (do-person-location 'person)
                            (do-person-location 'location))))
```

The output-person-location function takes one parameter which is a list of responses that are in the same order as the global experimental data. It prints the comparison of that

data to the experimental results and prints a table of the response times for targets and foils.

```
(defun output-person-location (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *person-location-data*)
    (mean-deviation rts *person-location-data*)
    (format t "~%TARGETS:~%                          Person fan~%")
    (format t  "  Location      1             2             3~%")
    (format t "    fan")

    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* i 3)) data))))

    (format t "~%~%FOILS:")
    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* (+ i 3) 3)) data))))))
```

The grouped model is really just a demonstration of partial matching. The experiment code is only there to collect and display key presses of the model. It is not an interactive experiment which a person can also do nor does it have a direct comparison to any existing data.

```
(defvar *response* nil)

(defun run-grouped-recall ()
    (setf *response* nil)
    (reset)
    (run 20.0)
    (reverse *response*))

(defun record-response (value)
    (push value *response*))
```

The experiment code is very simple. First it defines a global variable *response* that will be used to hold the model's responses.

```
(defvar *response* nil)
```

The run-grouped-recall function takes no parameters. It clears the *response* list, runs the model, and then returns the list of responses in order. Because the responses are pushed onto the front of the list by the data collection function below it needs to be reversed before returning it.

```
 (defun run-grouped-recall ()
    (setf *response* nil)
    (reset)
    (pm-run 20.0)
    (reverse *response*))
```

The data collection is not done through pressing keys because there isn't an interface to accept them. Instead, the model simulates that process by directly calling a Lisp function

to record the response. How that is done will be described next. This is the function that is called, and all it does is push the response onto the global list.

```
(defun record-response (value)
    (push value *response*))
```

Calling Lisp code from within a production is something new in this model, and it's done in the harvest-first-item, harvest-second-item, and harvest-third-item productions. Here is the harvest-third-item production.

```
(p harvest-third-item
   =goal>
      isa      recall-list
      element  third
      group    =group
   =retrieval>
      isa      item
      name     =name
==>
   =goal>
      element  fourth
   +retrieval>
      isa      item
      parent   =group
      position fourth
      :recently-retrieved nil
   !eval! (record-response =name)
)
```

The new operation shown in that production is !eval!. [The '!' is called bang in Lisp, so that's pronounced bang-eval-bang.] It can be placed on either the LHS or RHS of a production and must be followed by a Lisp expression which will be evaluated.

On the RHS of a production all it does is evaluate an expression. When the production fires, the !eval! is just another action that occurs. Note that the expression can contain variables from the production and the current binding of that variable in the instantiation is what will be used in the expression.

However, on the LHS of a production a !eval! specifies a condition that must be met before the production can be selected as with all other LHS conditions. The value returned by the evaluation of a LHS !eval! call must be non-nil for the production to be selected. Because it will be called during the selection process, a LHS !eval! is likely to be evaluated very often and will be called even when the production that it is in is not the one that will be eventually selected and fired.

!eval! is a powerful tool, but one that can easily be abused. Using it to call Lisp as an abstraction for an aspect of the model which is not necessary to model in detail for a particular task is the recommended use. In general, the predictions of the model should not depend heavily on the use of !eval!, otherwise there is not really any point to using ACT-R to model – you might as well just generate some functions to produce the data you want. !eval! should **not** be used in writing the assignment models for the tutorial units.

In this model it is used to collect the model's responses without needing the overhead of creating an experiment with which to interact. Because this task is not concerned with response time or stimuli acquisition there is not really a need for an interactive experiment because it would only serve to increase the size and complexity of the model without adding anything to its purpose.

The **Siegler** model also contains a "model only" experiment because there is no display to see and it records the model's speech as a response. There is only one new function used, so it should be easy to follow.

Here is the experiment code:

```
(defvar *response*)

(defvar *siegler-data* '((0   .05 .86  0  .02  0  .02  0   0  .06)
                         (0   .04 .07 .75 .04  0  .02  0   0  .09)
                         (0   .02  0  .10 .75 .05 .01 .03  0  .06)
                         (.02  0  .04 .05 .80 .04  0  .05  0   0)
                         (0    0  .07 .09 .25 .45 .08 .01 .01 .06)
                         (.04  0   0  .05 .21 .09 .48  0  .02 .11))
  "The experimental data to be fit")

(defun test-fact (arg1 arg2)
  (reset)
  (install-device (open-exp-window "" :visible nil))
  (new-digit-sound arg1)
  (new-digit-sound arg2 .75)
  (setf *response* nil)
  (run 30)
  *response*)

(defmethod device-speak-string ((win rpm-window) text)
  (setf *response* text))

(defun do-siegler-set ()
  (list (test-fact 1 1)
        (test-fact 1 2)
        (test-fact 1 3)
        (test-fact 2 2)
        (test-fact 2 3)
        (test-fact 3 3)))


(defun run-siegler (n)
  (let ((responses nil))
    (dotimes (i n)
      (push (do-siegler-set) responses))
    (analyze responses)))

(defun analyze (responses)
  (display-results
   (mapcar (lambda (x)
             (mapcar (lambda (y)
                       (/ y (length responses))) x))
     (apply #'mapcar
            (lambda (&rest z)
              (let ((res nil))
                (dolist (i '("zero" "one" "two" "three"
                             "four" "five" "six" "seven" "eight"))
```

```
                  (push (count i z :test #'string-equal) res)
                  (setf z (remove i z :test #'string-equal)))
                (push (length z) res)
                (reverse res)))
            responses))))

(defun display-results (results)
  (let ((questions '("1+1" "1+2" "1+3" "2+2" "2+3" "3+3")))
    (correlation results *siegler-data*)
    (mean-deviation results *siegler-data*)
    (format t "    0     1     2     3     4     5     6     7     8   Other~%")
    (dotimes (i 6)
      (format t "~a~{~6,2f~}~%" (nth i questions) (nth i results)))))
```

It starts by defining a global variable to hold the response.

```
(defvar *response*)
```

Then it defines a global variable that holds the experimental data to be fit.

```
(defvar *siegler-data* '((0   .05 .86  0  .02  0  .02  0   0  .06)
                         (0   .04 .07 .75 .04  0  .02  0   0  .09)
                         (0   .02  0  .10 .75 .05 .01 .03  0  .06)
                         (.02  0  .04 .05 .80 .04  0  .05  0   0)
                         (0    0  .07 .09 .25 .45 .08 .01 .01 .06)
                         (.04  0   0  .05 .21 .09 .48  0  .02 .11))
  "The experimental data to be fit")
```

The test-fact function takes two parameters which are the numbers to present.

```
(defun test-fact (arg1 arg2)
```

The model is reset.

```
  (reset)
```

A virtual experiment window is created and installed for the model.

```
  (install-device (open-exp-window "" :visible nil))
```

Events are created to present the numbers aurally to the model.

```
  (new-digit-sound arg1)
  (new-digit-sound arg2 .75)
```

The *response* variable is cleared.

```
  (setf *response* nil)
```

Then the model is run to generate a response.

```
  (run 30)
```

The value of the *response* variable is returned.

```
  *response*)
```

The device-speak-string method will store the models vocal output in the *response* variable.

```
(defmethod device-speak-string ((win rpm-window) text)
  (setf *response* text))
```

The do-siegler-set function takes no parameters. It runs one trial of each of the 6 stimuli and returns the list of the responses.

```
(defun do-siegler-set ()
  (list (test-fact 1 1)
        (test-fact 1 2)
        (test-fact 1 3)
        (test-fact 2 2)
        (test-fact 2 3)
        (test-fact 3 3)))
```

The run-siegler function takes one parameter which is the number of times to repeat the set of 6 test stimuli. It runs that many times over the set of stimuli and then outputs the results.

```
(defun run-siegler (n)
  (let ((responses nil))
    (dotimes (i n)
      (push (do-siegler-set) responses))
    (analyze responses)))
```

The analyze function takes one parameter which is a list of lists where each sublist contains the responses to the 6 test stimuli. It calculates the percentage of each of the answers from zero to eight or other from the given trial data and calls display-results to print out that information.

```
(defun analyze (responses)
  (display-results
   (mapcar (lambda (x)
             (mapcar (lambda (y)
                       (/ y (length responses))) x))
     (apply #'mapcar
            (lambda (&rest z)
              (let ((res nil))
                (dolist (i '("zero" "one" "two" "three"
                             "four" "five" "six" "seven" "eight"))
                  (push (count i z :test #'string-equal) res)
                  (setf z (remove i z :test #'string-equal)))
                (push (length z) res)
                (reverse res)))
            responses))))
```

The display-results function takes one parameter which is a list of six lists where each sublist is a list of nine items which represents the percentages of the answers 0-8 or other for each of the 6 test stimuli. It prints out the comparison to the experimental data and then displays the table of the results.

```
(defun display-results (results)
  (let ((questions '("1+1" "1+2" "1+3" "2+2" "2+3" "3+3")))
    (correlation results *siegler-data*)
    (mean-deviation results *siegler-data*)
    (format t "    0     1     2     3     4     5     6     7     8   Other~%")
```

```
    (dotimes (i 6)
      (format t "~a~{~6,2f~}~%" (nth i questions) (nth i results)))))
```

Next, we'll look at the 1-hit blackjack model. It has a lot of Lisp code to go with it to play the game and control the opponent, and this code is flexible in that it allows for the game and opponent to be changed without having to change the existing code. How exactly to use that will be discussed in a separate section below, and for now we will just describe the code that's there.

Here's the experiment code:

```
(defvar *deck1* 'regular-deck)
(defvar *deck2* 'regular-deck)

(defun play-hands (hands &optional (print-game nil))
  (let ((scores (list 0 0 0 0)))
   (dotimes (i hands)
      (let* ((mcards (deal *deck1*))
             (ocards (deal *deck2*))
             (mchoice (show-model-cards (butlast mcards) (first ocards)))
             (ochoice (show-opponent-cards (butlast ocards) (first mcards))))

        (unless (string-equal "h" mchoice) (setf mcards (butlast mcards)))
        (unless (string-equal "h" ochoice) (setf ocards (butlast ocards)))

        (let* ((mtot (score-cards mcards))
               (otot (score-cards ocards))
               (mres (compute-outcome mcards ocards))
               (ores (compute-outcome ocards mcards)))

          (show-model-results mcards ocards mres ores)

          (when print-game
            (format t
              "Model: ~{~2d ~} -> ~2d (~4s)   Opponent: ~{~2d ~}-> ~2d (~4s)~%"
              mcards mtot mres ocards otot ores))

          (setf scores (mapcar #'+ scores
                               (list (if (eq mres 'win) 1 0)
                                     (if (eq ores 'win) 1 0)
                                     (if (and (eq mres 'bust) (eq ores 'bust)) 1 0)
                                     (if (and (= mtot otot) (not (eq mres 'bust))
                                              (not (eq ores 'bust))) 1 0)))))))
          scores))

(defun run-blocks (blocks block-size)
  (let (res)
    (dotimes (i blocks (reverse res))
      (push (play-hands block-size) res))))

(defun show-learning (n &optional (graph t) (game 'game0))
  (let ((data nil))
    (dotimes (i n)
      (reset)
      (funcall game)
      (if (null data)
          (setf data (run-blocks 20 5))
        (setf data (mapcar (lambda (x y)
                             (mapcar #'+ x y))
                           data
                           (run-blocks 20 5)))))
```

```
    (let ((percentages (mapcar (lambda (x) (/ (car x) (* n 5.0))) data)))
      (when graph
        (draw-graph percentages))
      (list (list (/ (apply #'+ (subseq percentages 0 5)) 5)
                  (/ (apply #'+ (subseq percentages 5 10)) 5)
                  (/ (apply #'+ (subseq percentages 10 15)) 5)
                  (/ (apply #'+ (subseq percentages 15 20)) 5))
                 percentages))))

(defun draw-graph (points)
  (open-exp-window "Data" :width 550 :height 430 :visible t)
  (add-line-to-exp-window '(50 0) '(50 400) :color 'white)
  (dotimes (i 5)
    (add-text-to-exp-window :x 0 :y (- 380 (* i 80)) :width 40
                            :text (format nil "~2,1f" (* (+ i 2) .1)))
    (add-line-to-exp-window (list 50 (- 390 (* i 80)))
                            (list 550 (- 390 (* i 80))) :color 'white))

  (let ((x 50))
    (mapcar (lambda (a b)
              (add-line-to-exp-window
                (list x (floor (- 550 (* 800 a))))
                (list (incf x 25) (floor (- 550 (* 800 b))))
                :color 'blue))
      (butlast points) (cdr points))))

(defun deal (deck)
  (list (funcall deck)
        (funcall deck)
        (funcall deck)))

(defun score-cards (list &optional (bust 21))
  (if (find 1 list)
      (special-score list bust)
    (apply #'+ list)))

(defun special-score (list bust)
  (let ((possible (list (apply #'+ list))))
    (dotimes (i (count 1 list))
      (push (+ (* 10 (1+ i)) (apply #'+ list)) possible))
    (apply 'max (remove-if (lambda (x) (> x bust)) possible))))

(defun compute-outcome (p1cards p2cards &optional (bust 21))
  (let ((p1tot (score-cards p1cards))
        (p2tot (score-cards p2cards)))
    (if (> p1tot bust)
        'bust
      (if (or (> p2tot bust) (> p1tot p2tot))
          'win
        'lose))))


(defun show-model-cards (mcards ocard)
  (if (buffer-read 'goal)
      (mod-focus-fct `(mc1 ,(first mcards) mc2 ,(second mcards) mc3 nil
                       mtot nil mstart ,(score-cards mcards)
                       mresult nil oc1 ,ocard oc2 nil oc3 nil otot nil
                       ostart ,(score-cards (list ocard))
                           oresult nil state start))
    (goal-focus-fct (car (define-chunks-fct
                           `((isa game-state mc1 ,(first mcards)
                                  mc2 ,(second mcards) mc3 nil
                                  mtot nil mstart ,(score-cards mcards)
                                  mresult nil oc1 ,ocard
                                  oc2 nil oc3 nil otot nil
                                  ostart ,(score-cards (list ocard))
                                  oresult nil state start))))))
```

```
  (setf *model-action* nil)
  (run-full-time 10)
  *model-action*
  )

(defvar *model-action* nil)

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *model-action* (string key)))

(defun show-model-results (mcards ocards mres ores)
  (if (buffer-read 'goal)
      (mod-focus-fct `(mc1 ,(first mcards)  mc2 ,(second mcards)
                       mc3 ,(third mcards) mtot ,(score-cards mcards)
                       mstart ,(score-cards (subseq mcards 0 2)) mresult ,mres
                       oc1 ,(first ocards) oc2 ,(second ocards)
                       oc3 ,(third ocards) otot ,(score-cards ocards)
                       ostart ,(score-cards (list (first ocards))) oresult ,ores
                       state results))
    (goal-focus-fct (car (define-chunks-fct
                          `((isa game-state mc1 ,(first mcards)
                                mc2 ,(second mcards) mc3 ,(third mcards)
                                mtot ,(score-cards mcards)
                                mstart ,(score-cards (subseq mcards 0 2))
                                mresult ,mres
                                oc1 ,(first ocards) oc2 ,(second ocards)
                                oc3 ,(third ocards)
                                otot ,(score-cards ocards)
                                ostart ,(score-cards (list (first ocards)))
                                oresult ,ores
                                state results))))))
  (run-full-time 10))


(defun regular-deck ()
  (min 10 (1+ (act-r-random 13))))

(defvar *opponent-rule* 'fixed-threshold)

(defun show-opponent-cards (cards mc1)
  (funcall *opponent-rule* cards mc1))

(defvar *opponent-threshold* 15)

(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*) "h" "s"))


(defun game0 ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-threshold* 15)
  (setf *opponent-rule* 'fixed-threshold))


(defvar *card-list* nil)

(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit))


(defun load-stacked-deck ()
  (let* ((card1 (+ 5 (act-r-random 6)))
         (card2 (+ 5 (act-r-random 6)))
```

```
         (card4 (if (> (act-r-random 1.0) .5) 2 8))
         (card3 (if (= card4 2) 10 (- 21 (+ card1 card2))))
         (card5 10)
         (card6 (if  (= card4 2) 10 2)))
    (list card1 card2 card3 card4 card5 card6)))

(defun stacked-deck ()
  (cond (*card-list* (pop *card-list*))
        (t (setf *card-list* (load-stacked-deck))
           (pop *card-list*))))

(defun always-hit (cards mc1)
  "h")


(defun number-sims (a b)
  (when (and (numberp a) (numberp b))
    (- (/ (abs (- a b)) (max a b)))))
```

It starts by defining two variables which are used to hold the functions to call to deal cards (described in more detail in the section on how to modify the game).

```
(defvar *deck1* 'regular-deck)
(defvar *deck2* 'regular-deck)
```

The play-hands function runs the model against the opponent for a set number of hands and returns a list of the results. If the optional parameter is provided as non-nil then it will also print the details of each hand played.

```
(defun play-hands (hands &optional (print-game nil))
  (let ((scores (list 0 0 0 0)))
```

Loop over the given number of hands.

```
   (dotimes (i hands)
```

Deal the cards and get the players' choices.

```
      (let* ((mcards (deal *deck1*))
             (ocards (deal *deck2*))
             (mchoice (show-model-cards (butlast mcards) (first ocards)))
             (ochoice (show-opponent-cards (butlast ocards) (first mcards))))

        (unless (string-equal "h" mchoice) (setf mcards (butlast mcards)))
        (unless (string-equal "h" ochoice) (setf ocards (butlast ocards)))
```

Get the final hand totals and compute the results.

```
        (let* ((mtot (score-cards mcards))
               (otot (score-cards ocards))
               (mres (compute-outcome mcards ocards))
               (ores (compute-outcome ocards mcards)))
```

Show the results to the model so that it can learn from the outcome.

```
          (show-model-results mcards ocards mres ores)
```

Print the details if requested.

```
          (when print-game
            (format t
              "Model: ~{~2d ~} -> ~2d (~4s)   Opponent: ~{~2d ~}-> ~2d (~4s)~%"
              mcards mtot mres ocards otot ores))
```

Add the hand's result to the running total of results.

```
          (setf scores (mapcar #'+ scores
                               (list (if (eq mres 'win) 1 0)
                                     (if (eq ores 'win) 1 0)
                                     (if (and (eq mres 'bust) (eq ores 'bust)) 1 0)
                                     (if (and (= mtot otot) (not (eq mres 'bust))
                                              (not (eq ores 'bust))) 1 0)))))))
```

Return the list of scores.

```
          scores))
```

Run-blocks plays a number of blocks of a fixed size number of hands using play-hands and returns the list of the block results.

```
(defun run-blocks (blocks block-size)
  (let (res)
    (dotimes (i blocks (reverse res))
      (push (play-hands block-size) res))))
```

Show learning takes one required parameter, which is the number of games of 100 hands to play, and two optional parameters. The first optional parameter determines whether or not a graph of the model's average winning results will be drawn and the second specifies a function to call to initialize the variables which define the decks and the opponent (as described in the section on modifying the game).

```
(defun show-learning (n &optional (graph t) (game 'game0))
  (let ((data nil))
```

Play n rounds.

```
    (dotimes (i n)
```

Reset the model.

```
      (reset)
```

Call the function to initialize the game.

```
      (funcall game)
```

Run the 100 hands as 20 blocks of 5 and collect the results.

```
      (if (null data)
          (setf data (run-blocks 20 5))
        (setf data (mapcar (lambda (x y)
                             (mapcar #'+ x y))
                           data
                           (run-blocks 20 5)))))
```

Compute the percentage of model wins in groups of 5 and also in groups of 25 and return the lists of those results.

```
    (let ((percentages (mapcar (lambda (x) (/ (car x) (* n 5.0))) data)))
      (when graph
        (draw-graph percentages))
      (list (list (/ (apply #'+ (subseq percentages 0 5)) 5)
                  (/ (apply #'+ (subseq percentages 5 10)) 5)
                  (/ (apply #'+ (subseq percentages 10 15)) 5)
                  (/ (apply #'+ (subseq percentages 15 20)) 5))
                 percentages))))
```

Draw-graph takes a list of model win percentages and displays them in a graph built using an experiment window and a new ACT-R GUI function to draw lines in experiment windows.

```
(defun draw-graph (points)
  (open-exp-window "Data" :width 550 :height 430 :visible t)
  (add-line-to-exp-window '(50 0) '(50 400) :color 'white)
  (dotimes (i 5)
    (add-text-to-exp-window :x 0 :y (- 380 (* i 80)) :width 40
                            :text (format nil "~2,1f" (* (+ i 2) .1)))
    (add-line-to-exp-window (list 50 (- 390 (* i 80)))
                            (list 550 (- 390 (* i 80))) :color 'white))

  (let ((x 50))
    (mapcar (lambda (a b)
              (add-line-to-exp-window
                (list x (floor (- 550 (* 800 a))))
                (list (incf x 25) (floor (- 550 (* 800 b))))
                :color 'blue))
      (butlast points) (cdr points))))
```

The deal function takes a parameter which is a function to call to get cards from a deck and returns the list of the next three cards to be dealt.

```
(defun deal (deck)
  (list (funcall deck)
        (funcall deck)
        (funcall deck)))
```

Score-cards takes a list of card values and an optional maximum score (which defaults to 21). It returns the score for the list of cards and counts 1's as 11 if it does not cause the score to exceed the maximum.

```
(defun score-cards (list &optional (bust 21))
  (if (find 1 list)
      (special-score list bust)
    (apply #'+ list)))
```

Special-score is the function used by score-cards to actually compute the score for the cards.

```
(defun special-score (list bust)
  (let ((possible (list (apply #'+ list))))
    (dotimes (i (count 1 list))
      (push (+ (* 10 (1+ i)) (apply #'+ list)) possible))
    (apply 'max (remove-if (lambda (x) (> x bust)) possible))))
```

Compute-outcome takes two lists of cards representing two players' hands and returns the result for the first player – one of bust, win, or lose.

```
(defun compute-outcome (p1cards p2cards &optional (bust 21))
```

```
  (let ((p1tot (score-cards p1cards))
        (p2tot (score-cards p2cards)))
    (if (> p1tot bust)
        'bust
      (if (or (> p2tot bust) (> p1tot p2tot))
          'win
        'lose))))
```

Show-model-cards takes a list of the model's starting cards and the number representing the face-up card of the opponent. It modifies the chunk in the goal buffer if there is one with that information, or creates a new goal chunk if there is not and then runs the model for 10 seconds and returns the keypress which the model makes.

```
(defun show-model-cards (mcards ocard)
```

If there is a chunk in the goal buffer

```
  (if (buffer-read 'goal)
```

Then modify it with the current information

```
      (mod-focus-fct `(mc1 ,(first mcards) mc2 ,(second mcards) mc3 nil
                       mtot nil mstart ,(score-cards mcards)
                       mresult nil oc1 ,ocard oc2 nil oc3 nil otot nil
                       ostart ,(score-cards (list ocard))
                           oresult nil state start))
```

If not, then create a new chunk and make that the current goal chunk.

```
    (goal-focus-fct (car (define-chunks-fct
                           `((isa game-state mc1 ,(first mcards)
                                 mc2 ,(second mcards) mc3 nil
                                 mtot nil mstart ,(score-cards mcards)
                                 mresult nil oc1 ,ocard
                                 oc2 nil oc3 nil otot nil
                                 ostart ,(score-cards (list ocard))
                                 oresult nil state start))))))
```

Clear the variable which will be set to the model's keypress.

```
  (setf *model-action* nil)
```

Run the model for exactly 10 seconds.

```
  (run-full-time 10)
```

Return the keypress value.

```
  *model-action*
  )
```

Define the variable to hold the model's keypress.

```
(defvar *model-action* nil)
```

The key-event-handler function just stores the model's keypress in the *model-action* variable.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *model-action* (string key)))
```

The show-model-results function is similar to show-model-cards except that it modifies the goal chunk with information about the outcome of the players' actions.

```
(defun show-model-results (mcards ocards mres ores)
```

If there's a chunk in the goal buffer

```
  (if (buffer-read 'goal)
```

Then modify it with the details of the results

```
      (mod-focus-fct `(mc1 ,(first mcards)  mc2 ,(second mcards)
                       mc3 ,(third mcards) mtot ,(score-cards mcards)
                       mstart ,(score-cards (subseq mcards 0 2)) mresult ,mres
                       oc1 ,(first ocards) oc2 ,(second ocards)
                       oc3 ,(third ocards) otot ,(score-cards ocards)
                       ostart ,(score-cards (list (first ocards))) oresult ,ores
                       state results))
```

If not, then create a new goal chunk with the information.

```
  (goal-focus-fct (car (define-chunks-fct
                             `((isa game-state mc1 ,(first mcards)
                                    mc2 ,(second mcards) mc3 ,(third mcards)
                                    mtot ,(score-cards mcards)
                                    mstart ,(score-cards (subseq mcards 0 2))
                                    mresult ,mres
                                    oc1 ,(first ocards) oc2 ,(second ocards)
                                    oc3 ,(third ocards)
                                    otot ,(score-cards ocards)
                                    ostart ,(score-cards (list (first ocards)))
                                    oresult ,ores
                                    state results))))))
```

Run the model for 10 seconds.

```
  (run-full-time 10))
```

This section of the code contains the functions which implement the games provided. There are more details on how that works and how it can be changed at the end of this document.

Regular-deck returns a card value from a standard deck distribution.

```
(defun regular-deck ()
  (min 10 (1+ (act-r-random 13))))
```

Define a variable to hold the opponent's function and set it to the default opponent.

```
(defvar *opponent-rule* 'fixed-threshold)
```

The show-opponent-cards function is similar to the show-model-cards function. It gets the opponent's two cards and the model's face-up card which it passes to the current opponent function to determine if the opponent will hit or stay.

```
(defun show-opponent-cards (cards mc1)
  (funcall *opponent-rule* cards mc1))
```

The *opponent-threshold* variable is used by the default opponent to set its cutoff point for hitting.

```
(defvar *opponent-threshold* 15)
```

The fixed-threshold function implements the default opponent which hits if it has a total less than the threshold set with the *opponent-threshold* variable.

```
(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*) "h" "s"))
```

The game0 function sets up the decks and opponent to play the default game specified in the unit.

```
(defun game0 ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-threshold* 15)
  (setf *opponent-rule* 'fixed-threshold))
```

The *card-list* variable is used to implement a different deck of cards.

```
(defvar *card-list* nil)
```

The game1 function sets up a different deck of cards and opponent to play against the model as described in the section at the end of this document.

```
(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit))
```

The load-stacked-deck function creates a set of cards to deal to the model and the opponent with the opponent's face-up card indicating what the model needs to do to win.

```
(defun load-stacked-deck ()
  (let* ((card1 (+ 5 (act-r-random 6)))
         (card2 (+ 5 (act-r-random 6)))
         (card4 (if (> (act-r-random 1.0) .5) 2 8))
         (card3 (if (= card4 2) 10 (- 21 (+ card1 card2))))
         (card5 10)
         (card6 (if  (= card4 2) 10 2)))
    (list card1 card2 card3 card4 card5 card6)))
```

The stacked-deck function deals the cards off of the deck created by load-stacked-deck as needed.

```
(defun stacked-deck ()
  (cond (*card-list* (pop *card-list*))
        (t (setf *card-list* (load-stacked-deck))
           (pop *card-list*))))
```

The always-hit function always returns "h" and thus can be used as an opponent that always hits.

```
(defun always-hit (cards mc1)
  "h")
```

The number-sims function is used to specify the similarity values between numbers for the model. When it gets two numbers in its parameters it returns the similarity between them using the equation shown in the unit. For all other values of its parameters it returns nil which means that the default similarity calculation will occur.

```
(defun number-sims (a b)
  (when (and (numberp a) (numberp b))
    (- (/ (abs (- a b)) (max a b)))))
```

Now we'll describe the new ACT-R functions used in this unit, and then we'll describe how one can adjust the game and opponent for the 1-hit blackjack game. The new ACT-R functions used in this unit are pdisable, mod-chunk-fct, mod-focus-fct, chunk-slot-value-fct, buffer-read, define-chunks-fct, new-digit-sound and add-line-to-exp-window. I'm going to start with a brief discussion of the difference between functions and macros in Lisp and why ACT-R uses one or the other in certain places and the effect that it has (if you don't care to know that level of detail you can skip down to the descriptions of the new operators).

When calling a function in Lisp the parameters are evaluated first. A macro however does not evaluate its parameters (that's not the only difference, but probably the most significant for current purposes). Many of the ACT-R commands you've seen are actually macros e.g. chunk-type, add-dm, and p. They are macros so that you don't have to worry about quoting symbols or lists and other issues with Lisp syntax, but because they don't evaluate their parameters there are some things that you can't do with the macros (at least not without using an explicit call to eval or backquoting but I'm not going to discuss either of those). For example, say you have a variable called *number* and you'd like to create a chunk that has the value of *number* in one of its slots. The following is not going to work:

```
> (defvar *number* 3)
*NUMBER*
> (chunk-type test slot)
TEST
> (add-dm (foo isa test slot *number*))
(FOO)
```

The add-dm macro doesn't evaluate the variable *number* to get its value. Instead that will create a chunk that literally contains *number* as shown here (dm is another ACT-R macro that prints out a chunk from declarative memory given its name):

```
> (dm foo)
FOO
  ISA TEST
   SLOT  *NUMBER*
```

To allow modelers to do things like that most of the ACT-R macros have a corresponding function that does the same thing and the naming convention in the ACT-R code is to add "-fct" to the name for the functional form of the command. When using the functional

form of an ACT-R command you have to pay more attention to Lisp syntax and make sure that symbols are quoted and lists are constructed appropriately, and often the required parameters are a little different – things that do not need to be in a list for the macro version need to be in a list for the function. For example, add-dm-fct requires a list of lists as parameters instead of just an arbitrary number of lists. Here is the code that would generate the chunk as desired above:

```
> (add-dm-fct (list (list 'foo 'isa 'test 'slot *number*)))
(FOO)
> (dm foo)
FOO
  ISA TEST
   SLOT  3
```

That's not the only way to construct the lists (there are many ways one could accomplish the same thing) but should help to make the difference clear.

Now that I've covered that here are the details of the new commands.

pdisable – This command can be used to disable productions. It takes any number of production names as parameters and disables those productions in the current model. A disabled production cannot be selected or fired. Disabled productions will be enabled again once the model is reset, or if explicitly enabled using the corresponding **penable** command.

mod-chunk and **mod-chunk-fct –** This command modifies a chunk. The macro version requires a chunk name and then any number of slot and value pairs and the function requires a chunk name and then a list of slot and value pairs. Each of the slots specified for the chunk is given the corresponding value. It returns the name of the chunk which was modified. Here is an example of each being used on the chunk foo created above:

```
> (mod-chunk foo slot 5)
FOO
> (dm foo)
FOO
  ISA TEST
   SLOT  5

(FOO)
> (mod-chunk-fct 'foo (list 'slot 6))
FOO
> (dm foo)
FOO
  ISA TEST
   SLOT  6
```

One important thing to note is that although the example shown here modifies a chunk which is in the model's declarative memory that should not be done in practice. Only chunks outside of DM should be modified directly by the user e.g. chunks that are in buffers or chunks which have been created for other purposes using define-chunks (described below).

mod-focus and **mod-focus-fct –** This command works exactly like mod-chunk to modify a chunk, except that instead of specifying a particular chunk mod-focus will only modify the chunk which is currently in the goal buffer. Thus, only the list of modifications to be made is specified. Here is an example using each to modify a chunk that is in the goal buffer:

```
> (buffer-chunk goal)
GOAL: SIMPLE-GOAL0-0 [SIMPLE-GOAL0]
SIMPLE-GOAL0-0
  ISA SIMPLE-GOAL
   STATE  NIL

(SIMPLE-GOAL0-0)
> (mod-focus state next)
SIMPLE-GOAL0-0
> (buffer-chunk goal)
GOAL: SIMPLE-GOAL0-0
SIMPLE-GOAL0-0
  ISA SIMPLE-GOAL
   STATE  NEXT

(SIMPLE-GOAL0-0)
> (mod-focus-fct (list 'state 'busy))
SIMPLE-GOAL0-0
> (buffer-chunk goal)
GOAL: SIMPLE-GOAL0-0
SIMPLE-GOAL0-0
  ISA SIMPLE-GOAL
   STATE  BUSY

(SIMPLE-GOAL0-0)
```

chunk-slot-value and **chunk-slot-value-fct –** This command returns the value in a particular slot of a chunk. The parameters for both versions are the name of the chunk and the name of the slot. Here is an example of each again being used on the chunk foo from the mod-chunk example:

```
> (chunk-slot-value foo slot)
6

> (chunk-slot-value-fct 'foo 'slot)
6
```

buffer-read is a function (even though it does not have a –fct on the end). It takes one parameter which must be the name of a buffer and returns the name of the chunk currently in that buffer.

define-chunks and **define-chunks-fct –** This command creates new chunks just like add-dm and add-dm-fct. The macro version takes any number of lists of chunk descriptions and the function requires a list of chunk description lists. The difference between this and add-dm is that define-chunks does not add the newly created chunks to the declarative memory of the model. Often there is no need to put chunks directly into declarative memory if they are going to be placed into buffers (and thus end up there automatically anyway). In fact there are some times where doing so would be a problem for the model because it may then have a memory which can be retrieved of an event which it hasn't actually completed.

new-digit-sound – This command creates a new sound stimulus for the model to present numeric information and is similar to the new-tone-sound command which was used in unit 3 to present a tone. It requires one parameter which is the number to present and also takes an optional parameter to specify the time at which the digit should be presented. The current time will be used if a specific onset time is not given.

add-line-to-exp-window – this is similar to the add-text-to-exp-window function. It draws a line in the window that was opened with **open-exp-window**. It takes two required parameters and two keyword parameters. The required parameters specify the pixel coordinates of the end points of the line and each should be a two element list of the x and y coordinate. One keyword parameter is :color which can be used to specify the color that is used to draw the line. It must be a symbol (be careful especially when using ACL because many of these names are global variables that evaluate to an rgb object which is not the same as the symbol), and the colors which can be used are black, blue, red, green, white, pink, yellow, gray, light-blue, dark-green, purple, brown, light-gray, or dark-gray. The default is black if it is not provided. The other keyword parameter is :window which can be provided to indicate in which window to draw the line, but if there is only one open experiment window then it is not required.

Modifying 1-hit blackjack

The last section for this unit will be to discuss how to change the decks and/or the opponent for the 1-hit blackjack game as well as provide some suggestions for some other opponents to test a model against.

To make the game flexible the code relies on functions being specified to handle the three changeable components of the game: the model's deck of cards, the opponent's deck of cards, and the determination of whether or not the opponent will hit or stay. The functions are stored in global variables which are then called when needed. Thus, writing new functions for those parts of the game and setting the corresponding variables to those functions will change the way the game plays. To help make that more manageable, a fourth function can be written to set all the appropriate values for a particular game scenario and that function can be passed to the show-learning function as the (optional) third parameter. That setup function will be called at the start of each of the rounds that is played by show-learning. Thus, to play 5 rounds of a game specified by a function named new-game and display the graph of the model's results this call would be made to show-learning:

```
(show-learning 5 t 'new-game)
```

The functions for the decks are held in the variables *deck1* and *deck2*. The *deck1* function will be called to get cards for the model and *deck2* will be used for the opponent's cards. The deck function should return a number from 1-10 representing the value of the card being dealt. On every hand each of the functions will be called three

times. The first call will be for the face up card's value, the second call will be for the player's hidden card and the third call will be for the card that the player will receive on a hit. All three cards are dealt at the start of the hand, but only shown to the players when appropriate. The model's cards are dealt before the opponent's cards. Thus, if the same deck function is used for both players, as it is for the example games and the suggested alternatives, then the function will be called 6 times per hand with the first three calls returning the model's cards and the second set of three being for the opponent's cards.

For the default game described in the unit text both the model and opponent are dealt cards from this function:

```
(defun regular-deck ()
 (min 10 (1+ (act-r-random 13))))
```

which generates a value randomly from the appropriate distribution. The two deck variables are set to that function:

```
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
```

The function for the model's opponent is called to determine if the opponent will hit or stay. The opponent function is stored in the variable *opponent-rule* and will be passed two parameters. The first parameter is a list of the opponent's two starting cards. The second parameter is the number of the model's face up card. That function should return either the string "h" if the opponent will hit the hand or the string "s" if the opponent will stay for that hand. Because we are only using a fixed opponent for the model there is no function called to provide feedback on the outcome of the hand to the opponent i.e. it has no way to learn about the game or the model.

For the default game from the unit the opponent has a simple rule of always hitting when it has a total of less than 15 for its starting cards. This is the function which we have to represent that:

```
(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*) "h" "s"))
```

The score-cards function used there computes the score for a list of card values taking into account the rule of a 1 being counted as 11 when possible. Also note that we have introduced another variable for manipulating the game with the fixed-threshold function. The value at which the opponent will stay is set with the variable *opponent-threshold*. These variables are then set to create the default game scenario:

```
  (setf *opponent-threshold* 15)
  (setf *opponent-rule* 'fixed-threshold)
```

All of those variable settings are made in a function called game0:

```
(defun game0 ()
  (setf *deck1* 'regular-deck)
```

```
  (setf *deck2* 'regular-deck)
  (setf *opponent-threshold* 15)
  (setf *opponent-rule* 'fixed-threshold))
```

That game is the default value for the third parameter to show-learning if no game function is specified.

By writing a different game function to set the control variables one can easily modify the game that is played by the model. The simplest change would be to just adjust the threshold at which the default opponent stays. That could be done by adding a new game function to set the variables appropriately and then calling show-learning with that function's name as the third parameter. A function like this would change the opponent to stay when it has a score of 12 or more instead:

```
(defun new-game ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-threshold* 12)
  (setf *opponent-rule* 'fixed-threshold))
```

Then to run that game we would pass the name new-game to show learning like this:

```
(show-learning 5 t 'new-game)
```

There is a second game already programmed and available in the starting model. It is called game1 and the setup function looks like this:

```
(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit))
```

It sets a variable called *card-list* to nil, sets both players' decks to the same function, stacked-deck, and then sets the opponent to a function called always-hit. The operation of the opponent function should be fairly obvious from its name – it is an opponent who hits on every hand and the function looks like this:

```
(defun always-hit (cards mc1)
  "h")
```

Regardless of the cards in play the function returns "h" to indicate a hit.

The stacked-deck function is designed so that the opponent's face up card will be a perfect indicator to the model as to whether or not it should hit or stay to win the hand. This is designed as a test case for the model because it should be able to learn that and thus make the correct choice every time for the later hands in a round. Here is the definition of the stacked-deck function:

```
 (defun stacked-deck ()
```

```
  (cond (*card-list* (pop *card-list*))
        (t (setf *card-list* (load-stacked-deck))
           (pop *card-list*))))
```

It returns the next card stored in the *card-list* variable if there is one, otherwise it creates the new list of cards with the load-stacked-deck function and then returns the first of those. Here is the load-stacked-deck function:

```
(defun load-stacked-deck ()
  (let* ((card1 (+ 5 (act-r-random 6)))
         (card2 (+ 5 (act-r-random 6)))
         (card4 (if (> (act-r-random 1.0) .5) 2 8))
         (card3 (if (= card4 2) 10 (- 21 (+ card1 card2))))
         (card5 10)
         (card6 (if  (= card4 2) 10 2)))
    (list card1 card2 card3 card4 card5 card6)))
```

It generates a list of cards such that if the face up card of the opponent, the fourth card to be dealt, is a 2 (which will occur randomly half the time) then the model will win if it stays. If the opponent's face up card is an 8, which will occur the other half of the time, then the model will only win if it hits.

There are some other game scenarios which we have designed to test the model's ability to learn, but which have not been included in the code. These can be implemented as an additional exercise if you would like, and of course, you are also free to create game scenarios of your own design for testing. The testing scenarios we outline here all assume the same deck function will be used for both the model and the opponent to keep things simpler, but that is not required for all games since the two variables can be specified separately.

Game 2: In this game the deck consists of only cards numbered 7 and the opponent will always stay. In this game the model will always win if it hits and it should be able to learn that within the 100 games.

Game 3: The deck consists of only cards valued 8, 9, and 10 in equal proportions and again the opponent will always stay. The model should also learn to always stay because it will always lose if it hits. Staying on every hand will result in winning about 38.9% of the games.

Game 4: The deck consists of the cards 2, 4, 6, 8, and 10 being cycled in that order repeatedly. Thus the possible deals will be:

Model's cards	**Opponent's cards**
2 4 6	8 10 2
4 6 8	10 2 4
6 8 10	2 4 6
8 10 2	4 6 8
10 2 4	6 8 10

The opponent for this game is one which always hits. In this game the model should be able to learn the correct move to win the 4 which can be won out of the 5 possible hands (80% wins by the end).

Game 5: The deck consists of only the following possible triples each equally likely in any deal: (2 10 10) (4 9 9) (6 8 8) (8 8 5) (9 9 3) (10 10 1). These hands are designed so that if the player's initial score is small (12, 13 or 14) then a hit will always bust and if the initial score is large (16, 18, or 20) then a hit will always score 21 total. The opponent for this game will randomly hit or stay with equal probability. The optimal strategy for this game will result in winning about 54% of the hands.