Unit3 Model Code Description

The experiments in this unit are more involved than those of the last unit. The experiments are more complicated, and they involve collecting data and comparing it to existing experimental results. Many of the functions that were introduced in the last unit will be seen again (some of them always have to be there, like run) and there will be a few more introduced in this unit.

Because it is often necessary to run a model multiple times and average the results for data comparison, running the model quickly can be important. One thing that can greatly improve the time it takes to run the model (i.e. the real time that passes not the simulated time which the model experiences) is to have it interact with a virtual window instead of interacting with a real window on the computer. Virtual windows are an abstraction of a window interface that is internal to ACT-R. From the model's perspective there is no difference between a virtual window and a real window, but there is much less overhead involved with virtual windows because they exist entirely within Lisp. The only down side to using a virtual window is that you cannot see it, which can make debugging the model more difficult. To help with that, the tools that were introduced in the last unit for building and manipulating windows work exactly the same for real windows and virtual windows. All that is necessary to switch between them is one parameter when the window is opened. Thus, you can build the experiment and debug the model using a window that you can see. Then with one small change make the window virtual and run the model quickly over many runs. How to do that will be described below.

Here is the experiment code from the Sperling model.

```
(defvar *responses* nil)
(defvar *show-responses* nil)

(defvar *sperling-exp-data* '(3.03 2.40 2.03 1.50))

(defun do-sperling-trial (onset-time)

  (reset)

  (let* ((lis (permute-list '("B" "C" "D" "F" "G" "H" "J"
                              "K" "L" "M" "N" "P" "Q" "R"
                              "S" "T" "V" "W" "X" "Y" "Z")))
         (answers nil)
         (tone (act-r-random 3))
         (window (open-exp-window "Sperling Experiment"
                                  :visible t
                                  :width 300
                                  :height 300)))

    (dotimes (i 3)
      (dotimes (j 4)
        (let ((txt (nth (+ j (* i 4)) lis)))
          (when (= i tone)
            (push txt answers))
          (add-text-to-exp-window :text txt
                                  :width 40
                                  :x (+ 75 (* j 50))
                                  :y (+ 101 (* i 50))))))
    (install-device window)
    (new-tone-sound (case tone (0 2000) (1 1000) (2 500)) .5 onset-time)
    (schedule-event-relative (+ .9 (act-r-random .2)) 'clear-screen)

    (proc-display)
    (setf *responses* nil)
    (run 30 :real-time t)

    (when *show-responses*
      (format t "~%~%answers: ~S~%responses: ~S~%" answers *responses*))

    (compute-score answers)))


(defun compute-score (answers)
  (let ((score 0))
    (dolist (x answers score)
      (when (member x *responses* :test #'string-equal)
        (incf score)))))


(defun clear-screen ()
  (clear-exp-window)
  (proc-display))


(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (unless (string= key " ")
    (push (string key) *responses*)))


(defun report-data (data)
  (correlation data *sperling-exp-data*)
  (mean-deviation data *sperling-exp-data*)
  (print-results data))

(defun print-results (data)
  (format t "~%Condition    Current Participant   Original Experiment~%")
  (do ((condition '(0.00 0.15 0.30 1.00) (cdr condition))
       (temp1 data (cdr temp1))
       (temp2 *sperling-exp-data* (cdr temp2)))
      ((null temp1))
    (format t " ~4,2F sec.          ~6,2F                ~6,2F~%"
              (car condition) (car temp1) (car temp2))))


(defun run-block ()
  (let ((times (permute-list '(0.0 .15 .30 1.0)))
        (result nil))
    (dolist (x times)
      (push (cons x (do-sperling-trial x)) result))
    (sort result #'< :key #'car)))


(defun run-sperling (n)
  (let ((results (list 0 0 0 0)))
    (dotimes (i n)
      (setf results (mapcar #'+ results (mapcar #'cdr (run-block)))))
    (report-data (mapcar #'(lambda (x) (/ x n)) results))))
```

First, two global variables are defined. *responses* will hold the list of keys pressed by the participant and *show-responses* is used as a flag to indicate whether or not to print out the participant's responses every trial.

```
(defvar *responses* nil)
(defvar *show-responses* nil)
```

Next we define a global variable that holds the results of the original experiment so that the model's performance can be compared to it.

```
(defvar *sperling-exp-data* '(3.03 2.40 2.03 1.50))
```

The do-sperling-trial function takes one parameter which is the delay time at which to present the auditory cue in seconds. It presents the data and waits for the responses, and returns the number of letters correctly recalled in the target row.

```
(defun do-sperling-trial (onset-time)
```

First it resets ACT-R and randomizes a list of letters

```
  (reset)

  (let* ((lis (permute-list '("B" "C" "D" "F" "G" "H" "J"
                              "K" "L" "M" "N" "P" "Q" "R"
                              "S" "T" "V" "W" "X" "Y" "Z")))
```

then it creates a variable to hold the list of target letters,

```
         (answers nil)
```

randomly chooses which row to be the target,

```
         (tone (act-r-random 3))
```

and opens a window to do the task. The keyword parameters used when opening this window are new in this unit and will be described below in detail.

```
         (window (open-exp-window "Sperling Experiment"
                                  :visible t
                                  :width 300
                                  :height 300)))
```

Then it displays 3 rows of 4 letters recording the letters that are in the target row in the answers variable.

```
    (dotimes (i 3)
      (dotimes (j 4)
        (let ((txt (nth (+ j (* i 4)) lis)))
          (when (= i tone)
            (push txt answers))
          (add-text-to-exp-window :text txt
                                  :width 40
                                  :x (+ 75 (* j 50))
                                  :y (+ 101 (* i 50))))))
```

The model is told that the opened window is the one to interact with

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

A tone is scheduled to occur for the model at the time specified which lasts for .5 seconds and has a frequency determined by which row is to be recalled.

```
    (new-tone-sound (case tone (0 2000) (1 1000) (2 500)) .5 onset-time)
```

At a random time between .9 and 1.1 seconds the clear-screen function (which is defined below) will be called

```
    (schedule-event-relative (+ .9 (act-r-random .2)) 'clear-screen)
```

The model is told to process the display

```
    (proc-display)
```

The variable to hold the model's responses is cleared

```
    (setf *responses* nil)
```

The model is run for up to 30 seconds in real time

```
    (run 30 :real-time t)
```

If the *show-responses* variable is set it prints out the correct answers and the responses

```
    (when *show-responses*
      (format t "~%~%answers: ~S~%responses: ~S~%" answers *responses*))
```

Then it calls the compute-score function (defined below) to return the number of correct responses

```
    (compute-score answers)))
```

The compute-score function takes one parameter which is a list of the letters in the target row. It returns the number of those items that were given by the participant.

```
(defun compute-score (answers)
  (let ((score 0))
    (dolist (x answers score)
      (when (member x *responses* :test #'string-equal)
        (incf score)))))
```

The clear-screen function is called at the appropriate time when the model is performing the task. It clears the screen and makes the model reprocess it.

```
(defun clear-screen ()
  (clear-exp-window)
  (pm-proc-display))
```

The rpm-window-key-event-handler gets called automatically when a key is pressed and is passed the window in which the press occurred and the key that was pressed. For this task it pushes the keys onto the *responses* list unless the space bar is pressed.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (unless (string= key " ")
    (push (string key) *responses*)))
```

The report-data function takes one parameter which is a list that should represent average data of participants in the task. Those data are compared to the original experimental data and the correlation, mean deviation, and table of the results are printed.

```
(defun report-data (data)
  (correlation data *sperling-exp-data*)
  (mean-deviation data *sperling-exp-data*)
  (print-results data))
```

The print-results function takes one parameter which should be a list containing average data of participants in the task. Those data are printed in a table with the onset times and the original data.

```
(defun print-results (data)
  (format t "~%Condition    Current Participant   Original Experiment~%")
  (do ((condition '(0.00 0.15 0.30 1.00) (cdr condition))
       (temp1 data (cdr temp1))
       (temp2 *sperling-exp-data* (cdr temp2)))
      ((null temp1))
    (format t " ~4,2F sec.          ~6,2F                ~6,2F~%"
              (car condition) (car temp1) (car temp2))))
```

The run-block function takes no parameters. It runs one trial of the task at each of the 4 onset times randomly ordered. It returns a list of conses ordered by onset time, with the car of each cons being the onset time and the cdr being the number of correct responses.

```
(defun run-block ()
  (let ((times (permute-list '(0.0 .15 .30 1.0)))
        (result nil))
    (dolist (x times)
      (push (cons x (do-trial x)) result))
    (sort result #'< :key #'car)))
```

The run-sperling function takes one parameter which is the number of blocks of the experiment to run. A block is one trial at each of the 4 onset times. The results of those blocks are averaged together and then the comparison of that average data to the original experimental data is printed.

```
(defun run-sperling (n)
  (let ((results (list 0 0 0 0)))
    (dotimes (i n)
      (setf results (mapcar #'+ results (mapcar #'cdr (run-block)))))
    (report-data (mapcar #'(lambda (x) (/ x n)) results))))
```

The new ACT-R and miscellaneous modeling functions that are used in this model are:

Open-exp-window – this was introduced in the last unit. Here we see it getting passed keyword parameters that were not used previously. :height and :width specify the size of the window in pixels. The :visible parameter is the flag that determines whether a real or a virtual window is used. If :visible is **t** (the default value if it is not specified) then a real window is used. If :visible is **nil**, then a virtual window is used. There are two other parameters, :x and :y which can be used to specify the position of the upper left corner of the window on the main display.

New-tone-sound – This function takes 2 required parameters and a third optional parameter. The first parameter is the frequency of a tone to be presented to the model. The second is the duration of that tone in seconds. If the third parameter is specified then it indicates at what time the tone is to be presented, and if it is omitted then the tone is to be presented immediately. At the requested time a tone sound will be made available to the model's auditory module using the requested frequency and duration.

Schedule-event-relative – This function takes 2 required parameters and a number of additional parameters. It is used to schedule functions to be called during the running of the model. The first parameter specifies an offset from the current time at which the function should be called. The second parameter is the function to call. By scheduling functions to be called during the running of the model it is possible to have the experiment change without stopping the model to do so. More information on the scheduling functions can be found in the ACT-R reference manual.

Correlation – this function takes 2 required parameters which must be equal length 'collections' of numbers. The numbers can be in arrays or lists and they do not both need to be in the same format (one could be an array and the other a list). The only requirement is that they have the same number of numbers in them. This function then extracts those numbers and computes the correlation between the two sets of numbers. That correlation value is returned. There is a keyword parameter :output which defaults to t. When :output is t the correlation is printed to *standard-output*. If :output is a string, stream, or pathname then it is used to open a stream to which the results are written.

Mean-deviation – this function operates just like correlation, except that the calculation performed is the root mean square deviation between the data sets.

Now we will look at the subitizing experiment code.

```
(defvar *response* nil)
(defvar *response-time* nil)
(defvar *who*)

(defvar *subitizing-exp-data*
    '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58))


(defun subitize-trial (n &optional who)
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"
                                 :visible t
                                 :width 300
                                 :height 300
                                 :x 300
                                 :y 300)))
    (dolist (point points)
      (add-text-to-exp-window :text "x"
                              :width 10
                              :x (first point)
                              :y (second point)))
    (setf *response* nil)
    (setf *response-time* nil)


    (if (not (eq who 'human))
        (progn
          (setf who 'model)
          (reset)
          (install-device window)
          (proc-display )
          (run 30 :real-time t))
      (progn
        (let ((start-time (get-time nil)))
          (while (null *response*)
            (allow-event-manager window))
          (setf *response-time* (- *response-time* start-time)))))

    (let ((response (if (and (eq who *who*) *response*)
                        (read-from-string *response*)
                        -1)))
      (list (if (or (not (eq who *who*)) (null *response-time*))
                30.0
                (/ *response-time* 1000.0))
            (or (= response n)
                (and (= n 10) (= response 0)))))))

(defun subitize (&optional who)
  (let (result)
    (dolist (items (permute-list '(10 9 8 7 6 5 4 3 2 1)))
      (push (list items (subitize-trial items who)) result))
    (report-data (mapcar 'second (sort result '< :key 'car)))))


(defun report-data (data)
  (let ((rts (mapcar #'first data)))
    (correlation rts *subitizing-exp-data*)
    (mean-deviation rts *subitizing-exp-data*)
    (print-results data)))

(defun print-results (predictions)
  (format t "Items    Current Participant   Original Experiment~%")
  (dotimes (i (length predictions))
    (format t "~3d        ~5,2f  (~3s)              ~5,2f~%"
      (1+ i) (car (nth i predictions)) (second (nth i predictions))
      (nth i *subitizing-exp-data*))))

(defun generate-points (n)
   (let ((points nil))
    (dotimes (i n points)
      (push (new-distinct-point points) points))))

(defun new-distinct-point (points)
  (do ((new-point (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))
                  (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))))
      ((not (too-close new-point points)) new-point)))

(defun too-close (new-point points)
  (some #'(lambda (a) (and (< (abs (- (car new-point) (car a))) 40)
                           (< (abs (- (cadr new-point) (cadr a))) 40)))
        points))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *who* 'human)
  (setf *response-time* (get-time nil))
  (setf *response* (string key)))

(defmethod device-speak-string ((win rpm-window) text)
  (setf *who* 'model)
  (setf *response-time* (get-time))
  (setf *response* text))
```

First we define some global variables to hold the response, the time of that response, and an indication of whether it was a person or the model that made the response.

```
(defvar *response* nil)
(defvar *response-time* nil)
(defvar *who*)
```

Then we create a global variable that holds the data from the original experiment so that the model's performance can be compared to it.

```
(defvar *subitizing-exp-data*
    '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58))
```

The subitize-trial function takes one parameter which is the number of items to present, and an optional parameter which can be used to run a person through the task. It presents one trial to either the model or a person as needed and then returns a list of two items. The first item is the response time and the second item indicates whether or not the response was correct (t) or incorrect (nil).

```
(defun subitize-trial (n &optional who)
```

First it builds a list of n points using the generate-points function (defined below) and opens a window

```
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"
                                 :visible t
                                 :width 300
                                 :height 300
                                 :x 300
                                 :y 300)))
```

Then it draws an x at each of those random points

```
    (dolist (point points)
      (add-text-to-exp-window :text "x"
                              :width 10
                              :x (first point)
                              :y (second point)))
```

It clears the response variables

```
    (setf *response* nil)
    (setf *response-time* nil)
```

Now, if the model is doing the task it sets who to indicate that, resets the model, tells it which window to look at, makes it process the display, and then run for up to 30 seconds

```
    (if (not (eq who 'human))
        (progn
          (setf who 'model)
          (reset)
          (install-device window)
          (proc-display )
          (run 30 :real-time t))
```

If a person is doing the task

```
      (progn
```

Record the time that the trial starts

```
        (let ((start-time (get-time nil)))
```

Wait for a response

```
          (while (null *response*)
            (allow-event-manager window))
```

Set the response time to the difference between the start time and the time the response was recorded

```
          (setf *response-time* (- *response-time* start-time)))))
```

Then for either participant the return list is generated. The response is checked for correctness and the time is converted to seconds, or set to 30 seconds if there was no response.

```
    (let ((response (if (and (eq who *who*) *response*)
                        (read-from-string *response*)
                        -1)))
      (list (if (or (not (eq who *who*)) (null *response-time*))
                30.0
                (/ *response-time* 1000.0))
              (or (= response n)
                  (and (= n 10) (= response 0)))))))
```

The subitize function takes one optional parameter which can be specified as human to run a person. It presents each of the 10 possible conditions once in random order collecting the data. It then prints out the data and the comparison to the experimental results.

```
(defun subitize (&optional who)
  (let (result)
    (dolist (items (permute-list '(10 9 8 7 6 5 4 3 2 1)))
      (push (list items (subitize-trial items who)) result))
    (report-data (mapcar 'second (sort result '< :key 'car)))))
```

The report-data function takes one parameter which is a list of response lists as are returned by the subitize-trial function. It prints the comparison of the response times to the experimental data and then prints a table of the response times and correctness.

```
(defun report-data (data)
  (let ((rts (mapcar #'first data)))
    (correlation rts *subitizing-exp-data*)
    (mean-deviation rts *subitizing-exp-data*)
    (print-results data)))
```

The print-results function takes one parameter which is a list of response lists as are returned by the subitize-trial function. It prints a table of the response times and correctness along with the original data.

```
(defun print-results (predictions)
  (format t "Items    Current Participant   Original Experiment~%")
  (dotimes (i (length predictions))
    (format t "~3d        ~5,2f  (~3s)              ~5,2f~%"
      (1+ i) (car (nth i predictions)) (second (nth i predictions))
      (nth i *subitizing-exp-data*))))
```

The generate-points function takes 1 parameter which specifies how many points to generate. It returns a list of n randomly generated points (lists of x and y coordinates) to use for displaying the items. The points are generated such that they are not too close to each other and within the default experiment window size.

```
(defun generate-points (n)
   (let ((points nil))
    (dotimes (i n points)
      (push (new-distinct-point points) points))))
```

The new-distinct-point function takes one parameter which is a list of points. It returns a new point that is randomly generated within the default experiment window boundary that is not too close to any of the points on the list provided.

```
(defun new-distinct-point (points)
  (do ((new-point (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))
                  (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))))
      ((not (too-close new-point points)) new-point)))
```

The too-close function takes two parameters. The first is a point and the second is a list of points. It returns t if the first point is within 40 pixels in either the x or y direction of any of the points on the list, otherwise it returns nil.

```
(defun too-close (new-point points)
  (some #'(lambda (a) (and (< (abs (- (car new-point) (car a))) 40)
                           (< (abs (- (cadr new-point) (cadr a))) 40)))
        points))
```

The rpm-window-key-event-handler method is called automatically when a key is pressed. It records the time of the key press and the key that was pressed. In this task only a person is pressing keys (the model is speaking the response) and it also sets the *who* variable to indicate that this is the human response. There is one important thing to note about this function. The do-trial function is looping until the *response* variable is set when a person does the task. Because this function gets called by the system asynchronously, it is important to set the variable that is being used as the flag that indicates it is finished last. Otherwise the do-trial function may attempt to use the *response-time* variable before it gets set.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *who* 'human)
  (setf *response-time* (get-time nil))
  (setf *response* (string key)))
```

The device-speak-string method is very similar to the rpm-window-key-event-handler. It is called automatically whenever the model speaks. It is also passed two parameters which are the current experiment window for the model and the text the model is speaking. It sets the global variables to record the response and when it occurred.

```
(defmethod device-speak-string ((win rpm-window) text)
  (setf *who* 'model)
  (setf *response-time* (get-time))
  (setf *response* text))
```

The only new ACT-R function used in this model is get-time.

get-time – this function takes an optional parameter and it returns the current time in milliseconds. If the optional parameter is not specified, then the model's simulated time is returned. If the optional parameter is specified as true then the time returned is the model's simulated time. If the optional parameter is specified as **nil** then the time is taken from the internal Lisp timer using the command get-internal-real-time.

Buffer stuffing

Although it was not used in the models, the buffer stuffing mechanism was introduced in this unit text. It mentioned that one can change the default conditions that are checked to determine which item (if any) will be stuffed into the visual-location buffer. The function which does that is called **set-visloc-default.** The parameters that you pass to it are essentially the same as you would specify in a request to the visual-location buffer. Here are a couple of examples:

```
(set-visloc-default isa visual-location :attended new screen-x lowest)
(set-visloc-default isa visual-location screen-x current > screen-y 100
                                        < screen-y 230))
(set-visloc-default isa visual-location kind text color red width highest)
```

set-visloc-default – This command sets the conditions that will be used to select the visual location that gets buffer stuffed. When the screen is processed by the model (proc-display is called) if the visual-location buffer is empty a visual-location that matches the conditions specified by this command will be placed into the visual-location buffer. Essentially, what happens is that when proc-display gets called, if the visual-location buffer is empty, a +visual-location request is automatically executed using the slot tests set with set-visloc-default.

Speeding up the experiments

The experiment code provided for both of these tasks uses a real window for interaction and thus the model must be run in real time to interact with that window. In the main unit text it mentions that it is possible to instead use a virtual window for the model and then it does not have to be run in real time. To make the experiment window virtual the :visible keyword parameter provided to open-exp-window must be specified as **nil** instead of **t**. In the sperling task that call occurs in the do-sperling-trial function:

```
(defun do-sperling-trial (onset-time)

  (reset)

  (let* ((lis (permute-list '("B" "C" "D" "F" "G" "H" "J"
                              "K" "L" "M" "N" "P" "Q" "R"
                              "S" "T" "V" "W" "X" "Y" "Z")))
         (answers nil)
         (tone (act-r-random 3))
         (window (open-exp-window "Sperling Experiment"
                                  :visible t
                                  :width 300
                                  :height 300)))
```

and in the subitizing task it is in the subitize-trial function:

```
(defun subitize-trial (n &optional who)
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"
                                 :visible t
                                 :width 300
                                 :height 300
                                 :x 300
                                 :y 300)))
```

Changing the highlighted **t** to **nil** in those functions will make them virtual windows.

After doing that, the call to run the model in those same functions can be changed to not run the model in real time. Like the :visible flag those calls to run include a keyword parameter :real-time which is starts as being specified as **t** as shown in this call which is used in both experiments:

```
(run 30 :real-time t)
```

Changing that **t** to **nil** will allow the model simulation to run without being constrained by real time. Alternatively, the real-time parameter could be removed entirely because the default operation of the run command is to run in simulated time:

```
 (run 30)
```

After making those changes to the experiment code you will have to save the file and then load it before they will take effect.