

This document (and the similar documents for future units) will describe the Lisp code that controls the experiments and how ACT-R is interfaced to them. It is not necessary that you write the experiments for models in Lisp, but since ACT-R runs in Lisp it is far and away the easiest way to do it and there are tools provided with ACT-R/PM to make the task more manageable. It is not required that you use these tools (ACT-R/PM can process and manipulate windows that contain typical interface elements (text, edit boxes, and buttons) no matter how they are generated in MCL and ACL), but one advantage of the tools provided is that they work the same on different systems. Thus, your model will be able to run on any machine that is running ACT-R 5 even if it does not have a graphic display (ACT-R/PM has a virtual window abstraction built into it). The ACT-R/PM documentation contains additional information on creating displays for those requiring more advanced experiments.

Here is the experiment code from the **demo2** model (the contents of the **misc** and **command** windows):

Misc window:

```
(defvar *response* nil)

(defun do-experiment ()
  (if *actr-enabled-p*
      (do-experiment-model)
      (do-experiment-person)))

(defun do-experiment-person ()

  (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"))))
         (text1 (first lis))
         (window (open-exp-window "Letter recognition")))

    (add-text-to-exp-window :text text1 :x 125 :y 150)

    (setf *response* nil)

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

    *response*))

(defun do-experiment-model ()

  (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"))))
         (text1 (first lis))
         (window (open-exp-window "Letter recognition")))

    (add-text-to-exp-window :text text1 :x 125 :y 150)

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

```

(pm-proc-display)

(setf *response* nil)

(pm-run 10)

*response*))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response* (string key))
  (clear-exp-window)
  (when *actr-enabled-p* (pm-proc-display)))

```

Commands Window:

```

(sgp :v t)

(pm-set-params :real-time t :show-focus t)

(goal-focus goal)

(setf *actr-enabled-p* t)

```

Before getting into the code, there are some details about the structure of the code that should be addressed. First, for all the experiments in the tutorial there will be two functions that run the experiment, one for a human participant and one for the model. The names of those functions will end in `-person` and `-model` respectively. This is not necessary since most of the code is identical it could easily be written as one function with a few conditionals in it, but by separating it this way we keep it simpler for discussion by avoiding those conditionals. The main function that is called to run the experiment will call the appropriate function between the two. We also use the Lisp convention of placing `*`'s around global variables' names and always define them. Because ACT-R is also running in the same Lisp if we were to use a variable in the experiment that was already used in the ACT-R files there could be some serious problems with the model. By defining the variables you use in the model you will be given a warning if you redefine a variable that is already in use and thus can change the name before it results in problems.

We will start with the **command** window code. `Sgp`, `pm-set-params`, and `goal-focus` should be familiar from the unit 1 and unit 2 texts.

We turn on the verbose flag so that the trace is displayed.

```
(sgp :v t)
```

We set the system to run in real time and enable the display of the attention ring.

```
(pm-set-params :real-time t :show-focus t)
```

We place the chunk named `goal` into the goal buffer.

```
(goal-focus goal)
```

Then there is the line:

```
(setf *actr-enabled-p* t)
```

actr-enabled-p is a global variable defined in ACT-R that operates as a flag to the system to let it know whether a model or person is interacting with the interface. It can also be used by the experiment code (as we will show below). When it is set to t the system operates as if a model is doing the task and if it is set to nil it operates as if a person is doing it. It should always be set appropriately in the experiment when using the experiment tools provided.

Now we will consider the code in the **misc** window. First, we see that there is a global variable defined called ***response***:

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

This variable is going to be used to record the key pressed during the trial.

Then there is the function that is called to run the experiment – **do-experiment**:

```
(defun do-experiment ()
  (if *actr-enabled-p*
      (do-experiment-model)
      (do-experiment-person)))
```

All this function does is test the ***actr-enabled-p*** variable and call the appropriate function for performing the task.

Next is the function that runs a person through the task. This function utilizes several of the experiment building functions included in ACT-R (those in red below) and they will be discussed in detail.

```
(defun do-experiment-person ()

  (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"))))
    (text1 (first lis))
    (window (open-exp-window "Letter recognition")))

    (add-text-to-exp-window :text text1 :x 125 :y 150)

    (setf *response* nil)

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

    *response*))
```

What this function does is select a random letter from a list,

```
(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"))))  
      (text1 (first lis)))
```

open a window in which to do the task

```
(window (open-exp-window "Letter recognition"))
```

display the chosen letter in the window

```
(add-text-to-exp-window :text text1 :x 125 :y 150)
```

clear the response variable

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

wait for a person to press a key

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

and then return the key that was pressed.

```
*response*))
```

The experiment construction functions used are:

Permute-list – this function takes one parameter which must be a list and returns a randomly ordered copy of that list.

Open-exp-window – this function takes one required parameter which is the title for the window. It can also take several keyword parameters that control how the window is displayed. Those are described fully in the manual and will be introduced in later units as necessary. This function opens a window for performing an experiment and returns that window. If there is already an experiment window open with that title it clears its contents and brings it to the foreground. If there is not already an experiment window with that title it closes the previous experiment window if one exists and opens a new window with the requested title and brings it to the foreground.

Add-text-to-exp-window – this function draws a static text string on the window that was opened using **open-exp-window**. It takes a few keyword parameters. `:text` specifies the text string to display. `:x` and `:y` specify the pixel coordinate of the upper-left corner of the box in which the text is to be displayed, and there are 3 others that are not shown: `:height`, `:width` and `:color`. Height and width specify the size of the box in which to draw the text in pixels. The default value for `:height` is 20 and for `:width` is 75. Color specifies

in which color the text will be drawn and defaults to black (there is a limited set of colors which are supported see the experiment writing text for more details).

While – this is a looping construct. It takes an arbitrary number of parameters. The first parameter specifies the test condition, and the rest specify the body of the loop. The test is evaluated and if it returns anything other than nil all of the forms in the body are executed in order. This is repeated until the test returns nil. Thus, while the test is true (non-nil) the body is executed.

Allow-event-manager – this function takes one parameter, which must be a window of the experiment. It calls the appropriate function of the system to handle user interaction. Giving the system a chance to handle the user interactions is important because otherwise the rpm-window-key-event-handler method (described below) will never be called.

Next, comes the function to run the model through the task. Much of the code is exactly the same as for a person (the green text). There are a couple of extra steps necessary to interface the model to the experiment shown in red, and instead of waiting for a key press as was done for a person the model is run to produce a key press.

```
(defun do-experiment-model ()

  (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")))
        (text1 (first lis))
        (window (open-exp-window "Letter recognition")))

    (add-text-to-exp-window :text text1 :x 125 :y 150)

    (reset)
    (pm-install-device window)
    (pm-proc-display)

    (setf *response* nil)

    (pm-run 10)

    *response*))
```

The functions used to interface the model to the task are important, and will be seen in every experiment. What they do are:

Reset – this function call does the same thing as pressing the reset button in the environment. It returns the model to time 0 and sets the state of the parameters, working memory, and the productions to those specified in the model file the last time it was loaded.

Pm-install-device – this function takes one parameter which must be a window or device (a device is an abstract representation used by ACT-R/PM which can be used for more complicated interactions – the full details of their use are covered in the ACT-R/PM

manual). This tells the model which window (or device) it is interacting with. All of the models actions (key presses, mouse movement and mouse clicks) will be sent to this window and the contents of this window will be what the model can “see”.

Pm-proc-display – this function can take one keyword parameter which is not used here. It tells the model to process the display for visual information. This function makes the model “look” at the window. Whenever the window is changed you must call **pm-proc-display** again to make sure the model becomes aware of those changes. The re-encoding described in the unit can only happen after this function is called, and the bottom-up visual attention mechanism discussed in unit 4 (buffer stuffing) will also only occur when this function is called. The keyword parameter :clear if specified as t will cause the model to treat the window as all new items – everything there will be considered unattended.

Pm-run – this function takes one required parameter which is the time to run a model in seconds and a keyword parameter not used here. It runs the model just like pressing the run button in the environment. The model will run until either the requested amount of time passes, or there is nothing left for the model to do (no productions will fire and there are no pending actions that can change the state). If the keyword parameter called :full-time is specified as t, then the model will be advanced the entire amount of time requested even if there are no productions that can fire.

The final function defined in the **misc** window is this one:

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response* (string key))
  (clear-exp-window)
  (when *actr-enabled-p* (pm-proc-display)))
```

This method is automatically called by the system when a key press occurs in an experiment window by either a person or a model. It is passed two parameters, the window in which the key press occurred and the character representing the key that was pressed.

In this experiment it does the following:

Set the global variable ***response*** to a string containing the key pressed. It is put in a string because it is easier to compare strings ignoring case (as will be done in the unit assignment’s experiment)

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

erase the contents of the window

```
(clear-exp-window)
```

and then if the model is doing the task it makes the model look at the window

```
(when *actr-enabled-p* (pm-proc-display)))
```

The reason the window is cleared is to demonstrate the re-encoding of the vision module. The clearing is done with the experiment construction function `clear-exp-window`.

Clear-exp-window - this function takes no parameters. It removes all of the items that have been added to the experiment window that was opened with `open-exp-window`.

The code to present the assignment's experiment is very similar to the code for the **demo2** model. The only real difference is that more items are displayed and the response is checked for correctness. Here is the `do-experiment-model` function to show those differences.

```
(defun do-experiment-model ()

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

        (target (first letters))
        (foil (second letters))
        (window (open-exp-window "Letter difference"))
        (text1 foil)
        (text2 foil)
        (text3 foil))

    (reset)

    (pm-install-device window)

    (case (random 3)
      (0 (setf text1 target))
      (1 (setf text2 target))
      (2 (setf text3 target)))

    (add-text-to-exp-window :text text1 :x 125 :y 75)
    (add-text-to-exp-window :text text2 :x 75 :y 175)
    (add-text-to-exp-window :text text3 :x 175 :y 175)

    (setf *response* nil)

    (pm-proc-display)
    (pm-run 10)

    (if (string-equal *response* target)
        'correct
        nil)))
```

Here is the description of what it does.

Randomize the list of possible letters

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

        (target (first letters))

        (foil (second letters))

        (text1 foil)

        (text2 foil)

        (text3 foil))
```

pick one to be the target letter and one to be the foils

```
(target (first letters))
```

```
(foil (second letters))
```

open a window for the task

```
(window (open-exp-window "Letter difference"))
```

create variables to hold the text displayed and set them all to the foil

```
(text1 foil)
(text2 foil)
(text3 foil))
```

reset the model and tell it which window to interact with

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

randomly set one of the text items to the target letter

```
(case (random 3)
  (0 (setf text1 target))
  (1 (setf text2 target))
  (2 (setf text3 target)))
```

display the three letters

```
(add-text-to-exp-window :text text1 :x 125 :y 75)
(add-text-to-exp-window :text text2 :x 75 :y 175)
(add-text-to-exp-window :text text3 :x 175 :y 175)
```

clear the response variable

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

make the model look at the window

```
(pm-proc-display)
```

run the model

```
(pm-run 10)
```

compare the target letter to the model's response and return correct if they match and nil if they do not. The Lisp function string-equal is used because it is not case sensitive.

```
(if (string-equal *response* target)
    'correct
    nil)))
```