Unit2 Model Code Description

This document will describe the Lisp code that controls the experiments from unit 2 and how ACT-R is interfaced to them. One thing to note is that it is not necessary that one 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. There are tools provided with ACT-R which attempt to make the task more manageable when doing so, and these components are called the ACT-R GUI Interface (AGI). It is not required that you use these tools (ACT-R can process and manipulate windows that contain simple interface elements built in MCL, ACL and LispWorks automatically), but one advantage of the AGI is that it works the same on different systems. Thus, your model will be able to run on any machine that is running ACT-R 6.0 even if it does not have a graphic display because the AGI also works with a virtual window abstraction built into ACT-R.

Before getting into the specific code, there are some details about the structure of the code that should be addressed. For many of the experiments in the tutorial there will typically be one function that runs the experiment for either the model or a person. Most of the setup and control is the same regardless of whether it is a person or model doing the task, but the code necessary to actually "run" the model and person are different. To indicate which participant to run, most of the units will require specifying the symbol human for the function to run a person and will run the model otherwise. The code could also be written with two separate functions, one for a model and one for a human participant, but by using one function it is easier to see what pieces are the same and which differ.

Now we will look at the specific code. Here is the experiment code from the **demo2** model (everything that is outside of define-model except for the clear-all call):

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

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response* (string key))
  (clear-exp-window)
  (when *model*
    (proc-display)))

(defun do-demo2 (&optional who)

  (reset)

  (if (eq who 'human)
      (setf *model* nil)
    (setf *model* t))

  (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)
```

```
    (if *model*
        (progn
          (install-device window)
          (proc-display)
          (run 10 :real-time t))

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

    *response*))
```

Now, we will describe that code in detail and highlight the ACT-R and AGI functions used which will be described in detail at the end of this text.

First, it defines a global variable called ***response***:

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

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

Also define a variable called ***model*:**

```
(defvar *model* nil)
```

This variable will be set to indicate whether a person or model is performing the task.

Next we see the method (a function that is specific to a particular class of objects, in this case the rpm-window) that is automatically called by the system when a key press occurs in an experiment window regardless of where that key press came from (a model or a real participant). It is passed two parameters, the window in which the key press occurred and the character representing the key that was pressed.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
```

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))
```

Erases the contents of the window using a function from the AGI:

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

and then it makes sure that the model updates its visual representation of the window using the ACT-R command proc-display if the model is performing the task (as indicated by the *model* variable):

```
  (when *model*
   (proc-display)))
```

Next is the function that runs the experiment. It takes one optional parameter which can be used to specify that a person is doing the task:

```
(defun do-demo2 (&optional who)
```

The first thing it does is reset the ACT-R system. This is done for both the model and for real participants. It is important for the model so that all of its components are restored to their initial settings to prepare it to do the task. For a real participant this is only really necessary so that the seed parameter set with sgp in the task restores the initial seed for the pseudo-random number generator so that the same trial is generated every time:

```
  (reset)
```

Now it sets the value of *model* based on whether a person or model is performing the task:

```
(if (eq who 'human)
      (setf *model* nil)
    (setf *model* t))
```

Then it defines some local variables. The first one is a list of letters that are randomized with the AGI function permute-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")))
```

It then defines a variable called text1 with the first letter from that randomized list:

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

Then it defines a variable called window to hold the window returned from the AGI function open-exp-window which actually opens a window for use in the task:

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

Now it displays the letter in the window with an AGI function:

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

then it clears the *response* variable:

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

Now, depending on who the participant is (whether who is human or not) it performs the necessary steps to execute the task:

```
    (if *model*
```

If the model is doing the task then it must perform the following few actions (thus the need to group them with progn):

```
        (progn
```

First is must tell the model with what it should be interacting. In this case that is the window for the experiment:

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

Then the model is told to visually process that display:

```
          (proc-display)
```

Finally, the model is run for up to 10 seconds running in real time:

```
          (run 10 :real-time t))
```

If a person is doing the task then the function just waits for the *response* variable to change and calls the allow-event-manager AGI function while it waits to make sure that any real system events (OS or Lisp) that may be necessary are taken care of:

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

Finally, the function returns the value of the *response* variable:

```
    *response*))
```

The code to present the assignment's experiment is very similar to the code for the **demo2** model. The only real differences are that more items are displayed and the response is checked for correctness at the end. Here is its do-unit2 function with notes on the differences:

```
(defun do-unit2 (&optional who)

 (reset)

 (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")))
```

Define a variable called target to hold the different letter and one called foil to hold the letter that will be shown twice:

```
        (target (first letters))
        (foil (second letters))
```

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

Create three more variables that are all set to the foil letter for now:

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

Using the act-r-random function, randomly assign the target to one of the three letters:

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

Display all three letters and clear the response variable:

```
(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)
```

Just as in the previous task, if the model is performing the task install the device, have the model process the display, and then run for up to ten seconds, but if a person is doing the task just wait for a response to be made:

```
(if (not (eq who 'human))
    (progn
      (install-device window)
      (proc-display)
      (run 10 :real-time t))
  (while (null *response*)
    (allow-event-manager window)))
```

If the response matches the target letter then return correct otherwise return nil:

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

Here are more details on the ACT-R and AGI functions that were used. Because this is the first model that interacts with an experiment there are a lot of new functions to describe. Many of these will be used in almost all of the remaining models in the tutorial and later units will have fewer new functions introduced.

GUI creation and interaction

These functions are used to create windows and display information with which a model can interact effectively independent of the particular Lisp implementation (though

displaying windows with which a person interacts is dependent on using either the ACT-R environment or one of the supported Lisps).

Open-exp-window – this function takes one required parameter which is the title for the window which can be a string or symbol. It can also take several keyword parameters that control how the window is displayed 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 then it opens a new window with the requested title and brings it to the foreground. More than one experiment window can be open at a time. Any windows created with open-exp-window will be closed automatically when ACT-R is initialized by calling clear-all.

Add-text-to-exp-window – this function draws a static text string in a 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 4 others that are not used here :height, :width, :color, and :window. 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). :window must be specified if there has been more than one window opened in the current model to indicate which window to add the text to and can be provided either as the window object returned by open-exp-window or the title used when creating a window. If there is only one open window then it can be omitted as is the case here.

Clear-exp-window - this function has one optional parameter which if provided should specify a window. It removes all of the items that have been added to that window. If there is only one open window then the optional parameter is not needed.

Rpm-window-key-event-handler – this method can be defined by the modeler to process key presses that occur in the experiment window. The method must take two parameters. The first needs to be an instance of the rpm-window class. When an experiment window that has been opened with open-exp-window receives a key press (either from a model or a real user) it will pass the character that represents the key as the second parameter to this method and the first parameter will be the experiment window itself.

ACT-R Model Interaction and Setup

These are the functions that will be used over and over again for setting up and running the model.

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 productions to those specified in the define-model call, or reloads the file if there is no code in the define-model call.

Install-device – this function takes one parameter which must be a window or device (a device is an abstract representation of the world for ACT-R which can be used for more complicated interactions). 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".

Proc-display – this function can take one keyword parameter called clear (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 **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 3 (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.

Run – this function takes one required parameter which is the time to run a model in seconds and a keyword parameter called :real-time. 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 :real-time is specified as t, then the model is advanced in step with real time instead of being allowed to run as fast as possible in its own simulated time. That can be a useful thing to do when debugging a model or if it has to interact with software that is not designed to run with the model's simulated time. It is also possible for the modeler to specify an alternative clock for specifying a custom "real time", but that is beyond the scope of the tutorial.

Miscellaneous ACT-R/AGI Functions

These functions perform some tasks that can be useful when writing experiments for the model.

Act-r-random – this function operates like the ANSI Lisp function random except that it does not accept an optional random state because it uses a random state specific to the model. This allows models to perform identically on all Lisp platforms using a seed value that is easy to specify and independent of the pseudo-random number generator algorithm built into a particular Lisp. The algorithm used is the Mersenne Twister

generator which is considered to be among the best available for Monte Carlo simulations and is typically the same one that a Lisp implementation will use internally.

Permute-list – this function takes one parameter which must be a list and returns a randomly ordered copy of that list. It uses the act-r-random function to do so.

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 an experiment window. 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 may never be called for a real participant and the system may hang, unable to process user events.