Unit6 Model Code Description

The assignment for this unit is to write a complete experiment and model essentially from scratch. The demonstration experiments for this unit are not really useful examples for doing so. They are useful for showing how production parameter learning occurs, but because the experiments presented in this unit differ greatly from those of the previous units (and the one you will be writing) the discussion in this document is probably best avoided as an example of experiment generation for the assignment at hand.

In the Building Sticks Task (BST) there is not a simple response collected from the user, but instead an ongoing interaction that only ends when the correct results are achieved. This requires some new experiment generation functions and it is written in a mixture of the "trial at a time" and event-based styles. The individual trials are each a small event-based experiment, and the model is iterated over those trials one at a time.

Because both the **bst-learn** and **bst-no-learn** models use the same experiment (only the number of trials presented and the type of data collected differ) the discussion will focus on the **bst-learn** model.

```
(defvar *stick-a*)
(defvar *stick-b*)
(defvar *stick-c*)
(defvar *target*)
(defvar *current-stick*)
(defvar *current-line*)
(defvar *done*)
(defvar *choice*)
(defvar *experiment-window* nil)
(defvar *visible* nil)

(defvar *bst-exp-data* '(20.0 67.0 20.0 47.0 87.0 20.0 80.0 93.0
                         83.0 13.0 29.0 27.0 80.0 73.0 53.0))

(defparameter *bst-stimuli* '((15  250  55  125)(10  155  22  101)
                              (14  200  37  112)(22  200  32  114)
                              (10  243  37  159)(22  175  40  73)
                              (15  250  49  137)(10  179  32  105)
                              (20  213  42  104)(14  237  51  116)
                              (12  149  30  72)
                              (14  237  51  121)(22  200  32  114)
                              (14  200  37  112)(15  250  55  125)))

(defun build-display (a b c target)
    (setf *experiment-window* (open-exp-window "Building Sticks Task"
                                               :visible *visible*
                                               :width 600
                                               :height 400))

   (setf *stick-a* a)
   (setf *stick-b* b)
   (setf *stick-c* c)
   (setf *target* target)
   (setf *current-stick* 0)
   (setf *done* nil)
   (setf *choice* nil)
   (setf *current-line* nil)

   (add-button-to-exp-window :x 5 :y 23  :height 24 :width 40 :text "A"
```

```
                              :action #'button-a-pressed)
   (add-button-to-exp-window :x 5 :y 48  :height 24 :width 40 :text "B"
                              :action #'button-b-pressed)
   (add-button-to-exp-window :x 5 :y 73  :height 24 :width 40 :text "C"
                              :action #'button-c-pressed)
   (add-button-to-exp-window :x 5 :y 123 :height 24 :width 65 :text "Reset"
                              :action #'reset-display)

   (add-line-to-exp-window (list 75 35)  (list (+ a 75) 35) :color 'black)
   (add-line-to-exp-window (list 75 60)  (list (+ b 75) 60) :color 'black)
   (add-line-to-exp-window (list 75 85)  (list (+ c 75) 85) :color 'black)
   (add-line-to-exp-window (list 75 110) (list (+ target 75) 110) :color 'green)

   (allow-event-manager *experiment-window*))


(defun button-a-pressed (button)
   (declare (ignore button))

   (unless *choice* (setf *choice* 'under))

   (unless *done*
     (if (> *current-stick* *target*)
         (setf *current-stick* (- *current-stick* *stick-a*))
       (setf *current-stick* (+ *current-stick* *stick-a*)))
     (update-current-line)))

(defun button-b-pressed (button)
   (declare (ignore button))

   (unless *choice* (setf *choice* 'over))

   (unless *done*
     (if (> *current-stick* *target*)
         (setf *current-stick* (- *current-stick* *stick-b*))
       (setf *current-stick* (+ *current-stick* *stick-b*)))
     (update-current-line)))

(defun button-c-pressed (button)
   (declare (ignore button))

   (unless *choice* (setf *choice* 'under))

   (unless *done*
     (if (> *current-stick* *target*)
         (setf *current-stick* (- *current-stick* *stick-c*))
       (setf *current-stick* (+ *current-stick* *stick-c*)))
     (update-current-line)))

(defun reset-display (button)
   (declare (ignore button))

   (unless *done*

     (setf *current-stick* 0)
   (update-current-line)))

(defun update-current-line ()

   (when *current-line*
     (remove-items-from-exp-window *current-line*))

   (if (= *current-stick* *target*)
       (progn
         (setf *done* t)
         (setf *current-line* (add-line-to-exp-window (list 75 135)
                                                      (list  (+ *target* 75) 135)
```

```
                                                    :color 'blue))
        (add-text-to-exp-window :x 180 :y 200 :width 50 :text "Done"))
     (if (zerop *current-stick*)
           (setf *current-line* nil)
        (setf *current-line* (add-line-to-exp-window (list 75 135)
                                                     (list (+ *current-stick* 75) 135)
                                                     :color 'blue))))

   (allow-event-manager *experiment-window*)

   (proc-display))

(defun do-experiment (sticks who)

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

  (apply #'build-display sticks)
  (install-device *experiment-window*)

  (if (eq who 'human)
      (wait-for-human)
    (progn
      (proc-display :clear t)
      (run 60))))

(defun wait-for-human ()
  (while (not *done*)
    (allow-event-manager *experiment-window*))
  (sleep 1))

(defun bst-set (&optional who)
   (let ((result nil))
     (reset)

     (dolist (stim *bst-stimuli*)
       (do-experiment stim who)
       (push *choice* result))
     (reverse result)))

(defun bst-experiment (n &optional who)
   (let ((result (make-list (length *bst-stimuli*) :initial-element 0))
         (p-values (list '(decide-over 0) '(decide-under 0)
                         '(force-over 0) '(force-under 0))))
     (dotimes (i n result)
       (setf result (mapcar #'+ result (mapcar #'(lambda (x)
                                                   (if (equal x 'over) 1 0))
                                               (bst-set who))))
       (setf p-values (mapcar #'(lambda (x)
                                  (list (car x)
                                        (+ (second x) (production-u-value (car x)))))
                              p-values)))

     (setf result (mapcar #'(lambda (x) (* 100.0 (/ x n))) result))

     (when (= (length result) (length *bst-exp-data*))
       (correlation result *bst-exp-data*)
       (mean-deviation result *bst-exp-data*))

     (format t "~%Trial ")

     (dotimes (i (length result))
       (format t "~8s" (1+ i)))

     (format t "~%  ~{~8,2f~}~%~%" result)
```

```
    (dolist (x p-values)
      (format t "~12s: ~6,4f~%" (car x) (/ (second x) n)))))

(defun production-u-value (prod)
   (caar (no-output (spp-fct (list prod :u)))))
```

Because the model will be using the mouse to press buttons the following call also occurs in the model to ensure that the models hand is on the virtual mouse at the start of a run instead of on the keyboard.

```
(start-hand-at-mouse)
```

Here is the detailed description of the experiment code. It starts off defining a bunch of global variables. The first five are used to hold the length of the named item in pixels.

```
(defvar *stick-a*)
(defvar *stick-b*)
(defvar *stick-c*)
(defvar *target*)
(defvar *current-stick*)
```

The next one holds the screen object of the current line.

```
(defvar *current-line*)
```

Done is set when the user's line is the same length as the target line.

```
(defvar *done*)
```

Choice indicates whether the initial selection was overshoot or undershoot.

```
(defvar *choice*)
```

This one holds the window that is being used for the task.

```
(defvar *experiment-window* nil)
```

Finally, visible will be set to indicate whether or not the window should be displayed based on who is doing the task:

```
(defvar *visible* nil)
```

This variable holds the experimental data from the original experiment (percent of overshoot as initial strategy on each trial).

```
(defvar *bst-exp-data* '(20.0 67.0 20.0 47.0 87.0 20.0 80.0 93.0
                         83.0 13.0 29.0 27.0 80.0 73.0 53.0))
```

These are the stick lengths to use for each trail in the proper trial order. Each list represents one trial and the numbers are the pixel lengths of the sticks a, b, c, and the target respectively.

```
(defparameter *bst-stimuli* '((15  250  55  125)(10  155  22  101)
                              (14  200  37  112)(22  200  32  114)
                              (10  243  37  159)(22  175  40  73)
                              (15  250  49  137)(10  179  32  105)
```

```
(20  213  42  104)(14  237  51  116)
(12  149  30  72)
(14  237  51  121)(22  200  32  114)
(14  200  37  112)(15  250  55  125)))
```

The build-display function takes four parameters which are the pixel lengths of the sticks. It opens a window, draws the initial display and adds the buttons which the user will use to perform the task.

```
(defun build-display (a b c target)
```

Open a window and save it in the global variable.

```
  (setf *experiment-window* (open-exp-window "Building Sticks Task"
                                             :visible *visible*
                                             :width 600
                                             :height 400))
```

Initialize all of the global variables.

```
  (setf *stick-a* a)
  (setf *stick-b* b)
  (setf *stick-c* c)
  (setf *target* target)
  (setf *current-stick* 0)
  (setf *done* nil)
  (setf *choice* nil)
  (setf *current-line* nil)
```

Add the buttons that the user will use to do the task. This is a new function and all of the parameters will be explained in the detailed description below. The most important one is the action parameter which specifies a function to be executed when the button is pressed.

```
  (add-button-to-exp-window :x 5 :y 23 :height 24 :width 40 :text "A"
                            :action #'button-a-pressed)
  (add-button-to-exp-window :x 5 :y 48 :height 24 :width 40 :text "B"
                            :action #'button-b-pressed)
  (add-button-to-exp-window :x 5 :y 73 :height 24 :width 40 :text "C"
                            :action #'button-c-pressed)
  (add-button-to-exp-window :x 5 :y 123 :height 24 :width 65 :text "Reset"
                            :action #'reset-display)
```

Draw the initial set of lines on the display.

```
  (add-line-to-exp-window (list 75 35)  (list (+ a 75) 35) :color 'black)
  (add-line-to-exp-window (list 75 60)  (list (+ b 75) 60) :color 'black)
  (add-line-to-exp-window (list 75 85)  (list (+ c 75) 85) :color 'black)
  (add-line-to-exp-window (list 75 110) (list (+ target 75) 110) :color 'green)
```

Call the system dependent event manager to give the window a chance to display.

```
  (allow-event-manager *experiment-window*))
```

The next three functions, button-a-pressed, button-b-pressed, and button-c-pressed are called automatically when the corresponding buttons are pressed because they were specified in the add-button-to-exp-window calls. They all get passed the button itself as a parameter, but do not

need it. All three of them do basically the same thing. If this is the user's first choice it saves whether it is the overshoot or undershoot strategy. Then it computes the new length of the current stick after the application of the chosen stick (either addition or subtraction as appropriate), and then calls update-current-line to display the new configuration.

```
(defun button-a-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'under))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-a*))
      (setf *current-stick* (+ *current-stick* *stick-a*)))
    (update-current-line)))

(defun button-b-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'over))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-b*))
      (setf *current-stick* (+ *current-stick* *stick-b*)))
    (update-current-line)))

(defun button-c-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'under))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-c*))
      (setf *current-stick* (+ *current-stick* *stick-c*)))
    (update-current-line)))
```

The reset-display function is called when the reset button of the task is pressed. It takes one parameter which will be the button object, but it does not need it. It sets the length of the current stick to 0 and calls update-current-line to redisplay the state.

```
(defun reset-display (button)
  (declare (ignore button))

  (unless *done*
    (setf *current-stick* 0)
  (update-current-line)))
```

The update-current-line function takes no parameters. It redraws the current line on the screen and sets the done flag if the current stick is the same length as the target.

```
(defun update-current-line ()
```

First remove the old line if there is one.

```
  (when *current-line*
    (remove-items-from-exp-window *current-line*))

  (if (= *current-stick* *target*)
```

If the target has been reached then set the done flag, draw the stick and display the done message.

```
      (progn
        (setf *done* t)
        (setf *current-line*
             (add-line-to-exp-window (list 75 135) (list  (+ *target* 75) 135)
                                     :color 'blue))
        (add-text-to-exp-window :x 180 :y 200 :width 50 :text "Done"))
```

If the target has not yet been reached, then check to see if it has been reset. If it has been reset then just set the current line to nil, otherwise display a new line of the correct length.

```
    (if (zerop *current-stick*)
          (setf *current-line* nil)
        (setf *current-line*
              (add-line-to-exp-window (list 75 135)
                                      (list (+ *current-stick* 75) 135)
                                      :color 'blue))))
```

Call the system event manager so that the display gets a chance to update so that a visible window updates properly (does not matter for the virtual windows).

```
  (allow-event-manager *experiment-window*)
```

Have the model reevaluate the display.

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

The do-experiment function takes two parameters which should be a list of stick lengths and an indication of whether a person or model is doing the task. First, it sets the *visible* variable to t if a person is doing the task or nil for a model. Then, it calls build-display to open a window and display those sticks and calls install-device to indicate which window to interact with for the model. Finally, it calls the appropriate function to run one trial depending on whether a person or the model is doing the task.

```
(defun do-experiment (sticks who)

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

  (apply #'build-display sticks)
  (install-device *experiment-window*)

  (if (eq who 'human)
```

If it is a person doing the task then call the wait-for-human function which is defined below.

```
       (wait-for-human)
```

If it is the model then have it process the new display and run for up to 60 seconds (the assumption being that the model can always complete a trial in 60 seconds or less).

```
    (progn
      (proc-display :clear t)
      (run 60))))
```

The wait-for-human function takes no parameters. It just loops waiting for the done flag to be set. Then it waits one second before returning to give the person a chance to see the word done displayed.

```
(defun wait-for-human ()
  (while (not *done*)
    (allow-event-manager *experiment-window*))
  (sleep 1))
```

The bst-set function takes an optional parameter indicating whether a person or the model is doing the task. It resets the model, and then iterates over the list of stimuli in the global list *bst-stimuli* and runs one trial of each recording which strategy was chosen first on each one. It returns the list of strategy choices.

```
(defun bst-set (&optional who)
   (let ((result nil))
     (reset)

     (dolist (stim *bst-stimuli*)
       (do-experiment stim who)
       (push *choice* result))
     (reverse result)))
```

The bst-experiment function takes one required parameter which is the number of times to run the full experiment and an optional parameter to indicate whether it is a person or the model doing the task. It runs the experiment the requested number of trials and computes the percentage of times overshoot was chosen first on each trial. It compares that to the experimental data and displays a table of the results. It also records the learned value of utility for the productions that are responsible for the model's choices and reports their average at the end as well.

```
(defun bst-experiment (n &optional who)
   (let ((result (make-list (length *bst-stimuli*) :initial-element 0))
         (p-values (list '(decide-over 0) '(decide-under 0)
                         '(force-over 0) '(force-under 0))))
     (dotimes (i n result)
       (setf result (mapcar #'+ result
                            (mapcar #'(lambda (x)
                                        (if (equal x 'over) 1 0))
                                    (bst-set who))))
       (setf p-values (mapcar #'(lambda (x)
                                  (list (car x)
                                     (+ (second x) (production-u-value (car x)))))
                              p-values)))

     (setf result (mapcar #'(lambda (x) (* 100.0 (/ x n))) result))

     (when (= (length result) (length *bst-exp-data*))
       (correlation result *bst-exp-data* :output *out-file*)
       (mean-deviation result *bst-exp-data* :output *out-file*))

     (model-output "~%Trial ")

     (dotimes (i (length result))
       (model-output "~8s" (1+ i)))

     (model-output "~%  ~{~8,2f~}~%~%" result)

     (dolist (x p-values)
       (model-output "~12s: ~6,4f~%" (car x) (/ (second x) n)))))
```

The production-u-value function takes one parameter which should be a production name. It returns the u value of that production (the true utility without the noise being added) without printing the information to the listener.

```
(defun production-u-value (prod)
   (caar (no-output (spp-fct (list prod :u)))))
```

The new commands used in this model are:

**add-button-to-exp-window –** this function is similar to the add-text-to-exp-window function that you have seen many times before. It places a button in a window that was opened using **open-exp-window**. It takes a few keyword parameters. :text specifies the text to display on the button. :x and :y specify the pixel coordinate of the upper-left corner of the button, :height and :width specify the size of the button in pixels, and the :action parameter specifies a function to be called when this button is pressed. That function will be called with the button object itself as the only parameter. There is also a keyword parameter :window which can be used to indicate which window to place the button into when there is more than one window opened by open-exp-window.

**remove-items-from-exp-window –** this function takes any number of parameters and optionally the keyword parameter :window. Each item (other than the keyword value) must be an object that was added to an experiment window. Those objects are then removed from the specified window. If there is only one window open then the :window parameter does not need to be specified and the items will be removed from that window (as is the case here).

**no-output** – this command takes any number of forms. Those forms are evaluated with all of the ACT-R output commands disabled. This is often used to suppress the automatic printing from commands like sgp, sdp, spp, and dm which automatically print out details in addition to returning needed values. The return value from the no-output call is the value returned from the last form evaluated within it.