Unit 6 Code Description

The assignment for this unit is to write a complete experiment and model essentially from scratch. The demonstration experiment for this unit is much more complicated than the one needed for the assignment task. While it does provide information on creating more involved experiments for models, the models from earlier units (like the paired associate task in unit 4) will be more useful examples in completing the assignment for this unit.

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-driven styles. Each BST problem presented is a trial which is run as an event-driven experiment, and the model is iterated over those trials one at a time. The reason for using the event-driven approach for each problem is because the task is performed by pressing buttons and a button can have a function associated with it to call when it is pressed. The changes to the display can be performed by those functions as the model runs instead of having to stop it on each press, update the display, and then start running the model again. In this situation it is actually less complicated to write the event-driven task than it would be to build something that runs the model for each single action performed.

Lisp

Start by loading the model which can perform the task.

```
(load-act-r-model "ACT-R:tutorial;unit6;bst-model.lisp")
```

Create some global variables to hold the experiment information: the length of the target stick, the length of the current stick, the screen object for the current line, a flag for whether the task is complete, whether the person started with the over- or under- shoot strategy, the window for the experiment, and whether the window should be visible or virtual.

```
(defvar *target*)
(defvar *current-stick*)
(defvar *current-line*)
(defvar *done*)
(defvar *choice*)
(defvar *window* nil)
(defvar *visible* nil)
```

Some more global variables to hold the experimental data on choice of overshoot, the lengths of the sticks for the full experiment, and the lengths of the sticks for the 2 trial tests.

```
(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))

(defvar *exp-stims* '((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)))
```

```
(defvar *no-learn-stims* '((15  200  41 103)(10  200 29 132)))
```

The build-display function takes 4 parameters which are the lengths of the three given sticks and the length of the goal stick. It sets the global variables appropriately, opens a window, and then draws the starting information for the task.

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

  (setf *target* goal)
  (setf *current-stick* 0)
  (setf *done* nil)
  (setf *choice* nil)
  (setf *current-line* nil)
  (setf *window* (open-exp-window "Building Sticks Task" :visible *visible*
                                  :width 600 :height 400))
```

Add-button-to-exp-window is a new function for this unit and will be described in more detail at the end of this document. For now, the important thing to know is that when the button is created it can be associated with a command to call when it is pressed. The action parameter to the function can be a string which names a command or a list of a string which names a command and additional values. If it is just a command then that command will be called with no parameters when the button is pressed, and if it is a list then the command specified will be passed the other values in the list when the button is pressed. Thus, with the buttons created below, when the button labeled “Reset” is pressed it will call the “bst-reset-button-pressed” command with no parameters and when the button labeled “A” is pressed it will call the “bst-button-pressed” command with two parameters, the first being the length held in the variable a and the other being the symbol under.

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

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

The button-pressed function is associated with the A, B, and C buttons created above. It will be called when those buttons are pressed and passed the length of the corresponding stick and whether that button represents the over- or under- shoot strategy if it is the first one chosen.

```
(defun button-pressed (len dir)
```

If a previous strategy choice hasn’t been set, then set it to the current strategy.

```
  (unless *choice*
    (setf *choice* dir))
```

If the trial isn’t over yet then update the current stick using the chosen one.

```
  (unless *done*
```

```
    (if (> *current-stick* *target*)
        (decf *current-stick* len)
      (incf *current-stick* len))
    (update-current-line)))
```

The reset-display function gets called when the reset button is pressed and if the trial isn't over yet it will remove the current stick so the participant can start over.

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

Here we add the commands for the previous two functions so that they can be called as button actions.

```
(add-act-r-command "bst-button-pressed" 'button-pressed
                   "Choice button action for the Building Sticks Task.  Do not call directly")
(add-act-r-command "bst-reset-button-pressed" 'reset-display
                   "Reset button action for the Building Sticks Task.  Do not call directly")
```

Update-current-line redraws the current line and updates the global variables which hold the current state as necessary.

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

If the current stick has the target length the task is over. Set the done flag, modify the line to show its new length, and display the done prompt.

```
  (cond ((= *current-stick* *target*)
         (setf *done* t)
         (modify-line-for-exp-window *current-line* (list 75 135) (list (+ *target* 75) 135))
         (add-text-to-exp-window *window* "Done" :x 180 :y 200))
```

If the current stick is returned to length 0 then remove it from the screen and clear the variable holding the line.

```
        ((zerop *current-stick*)
         (when *current-line*
           (remove-items-from-exp-window *window* *current-line*)
           (setf *current-line* nil)))
```

If there is a current stick then update it to the new length.

```
        (*current-line*
          (modify-line-for-exp-window *current-line* (list 75 135)
                                      (list (+ *current-stick* 75) 135)))
```

Otherwise, draw a new current stick of the appropriate length and save that item.

```
        (t
         (setf *current-line* (add-line-to-exp-window *window* (list 75 135)
                                                      (list (+ *current-stick* 75) 135)
                                                      'blue)))))
```

The do-experiment function takes a list of lengths which represent the sticks to present on a trial and an optional parameter indicating whether a person will be performing the task. It calls build-display to create the window and then either waits for a person to finish the task or runs the model for up to 60 seconds to perform the task (in real time if the window is visible for the

model). When running the model, since this task requires using the mouse it first uses start-hand-at-mouse to put the model's hand on the virtual mouse instead of the virtual keyboard since the keyboard is the default location for the hands when using the experiment window device.

```
(defun do-experiment (sticks &optional human)
  (apply 'build-display sticks)
  (if human
      (when (visible-virtuals-available?)
        (wait-for-human))
    (progn
      (install-device *window*)
      (start-hand-at-mouse)
      (run 60 *visible*))))
```

The wait-for-human function loops until the task is over and then waits one second more after displaying the done prompt.

```
(defun wait-for-human ()
  (while (not *done*)
    (process-events))
  ;; wait for 1 second to pass after done
  (let ((start-time (get-time nil)))
    (while (< (- (get-time nil) start-time) 1000)
      (process-events))))
```

The bst-set function takes three required parameters which indicate whether a person is doing the task, whether the window should be visible, and a list of stick length lists for the trials to present. It also takes an optional parameter which indicates whether the model should be learning from trial to trial or not. It loops over the trials provided, resetting the model if it isn't supposed to learn from trial to trial, and records the initial strategy choice which was made on each one. It returns the list of strategy choices.

```
(defun bst-set (human visible stims &optional (learn t))
  (setf *visible* visible)
  (let ((result nil))
    (dolist (stim stims)
      (when (and (not human) (not learn))
        (reset))
      (do-experiment stim human)
      (push *choice* result))
    (reverse result)))
```

The bst-test function takes one required parameter which is the number of two-trial test runs to perform and an optional parameter indicating whether a person will be performing the task. It runs the required number of trials and counts the number of times over-shoot is used to solve them returning the list of those counts.

```
(defun bst-test (n &optional human)
  (let ((stims *no-learn-stims*))

    (let ((result (make-list (length stims) :initial-element 0)))
      (dotimes (i n result)
        (setf result (mapcar '+
                       result
                       (mapcar (lambda (x)
                                 (if (string-equal x "over") 1 0))
                         (bst-set human (or human (= n 1)) stims nil))))))))
```

The bst-experiment function takes one required parameter which is the number of times to perform the whole experiment and an optional parameter indicating whether a person will be performing the task. It runs the required number of experiments and computes the average proportion of times over-shoot is chosen for each trial and the average utility value of the critical productions in the model after each experiment. Those results are printed along with the correlation and mean-deviation from the original experimental data.

```
(defun bst-experiment (n &optional human)
  (let ((stims *exp-stims*))

    (let ((result (make-list (length stims) :initial-element 0))
          (p-values (list '(decide-over 0) '(decide-under 0) '(force-over 0) '(force-under 0))))
      (dotimes (i n result)
        (reset)
        (setf result (mapcar '+ result
                             (mapcar (lambda (x)
                                       (if (string-equal x "over") 1 0))
                               (bst-set human human stims))))
        (no-output
         (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))))))
```

The production-u-value function takes one parameter which should be the name of a production. It returns the current value of the :u parameter for that production.

```
(defun production-u-value (prod)
   (caar (spp-fct (list prod :u))))
```

Python

Start by importing the actr interface module and loading the model which can perform the task.

```
import actr

actr.load_act_r_model ("ACT-R:tutorial;unit6;bst-model.lisp")
```

Create some global variables to hold the experiment information: the length of the target stick, the length of the current stick, the screen object for the current line, a flag for whether the task is complete, whether the person started with the over- or under- shoot strategy, the window for the experiment, and whether the window should be visible or virtual.

```
target = None
current_stick = None
current_line = None
done = False
choice = None
window = None
visible = False
```

Some more global variables to hold the experimental data on choice of overshoot, the lengths of the sticks for the full experiment, and the lengths of the sticks for the 2 trial tests.

```
exp_data = [20, 67, 20, 47, 87, 20, 80, 93, 83, 13, 29, 27, 80, 73, 53]
exp_stims = [[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]]

no_learn_stims = [[15,200,41,103],[10,200,29,132]]
```

The build_display function takes 4 parameters which are the lengths of the three given sticks and the length of the goal stick. It sets the global variables appropriately, opens a window, and then draws the starting information for the task.

```
def build_display (a,b,c,goal):
    global window,target,current_stick,done,current_line,choice

    target = goal
    current_stick = 0
    done = False
    choice = None
    current_line = None
    window = actr.open_exp_window("Building Sticks Task",visible=visible,
                                  width=600,height=400)
```

Add_button_to_exp_window is a new function for this unit and will be described in more detail at the end of this document. For now, the important thing to know is that when the button is created it can be associated with a command to call when it is pressed. The action parameter to the function can be a string which names a command or a list of a string which names a command and additional values. If it is just a command then that command will be called with no parameters when the button is pressed, and if it is a list then the command specified will be passed the other values in the list when the button is pressed. Thus, with the buttons created below, when the button labeled “Reset” is pressed it will call the “bst-reset-button-pressed” command with no parameters and when the button labeled “A” is pressed it will call the “bst-button-pressed” command with two parameters, the first being the length held in the variable a and the other being the string under.

```
    actr.add_button_to_exp_window(window, text="A", x=5, y=23,
                                  action=["bst-button-pressed",a,"under"],
                                  height=24, width=40)
    actr.add_button_to_exp_window(window, text="B", x=5, y=48,
```

```
                                action=["bst-button-pressed",b,"over"],
                                height=24, width=40)
    actr.add_button_to_exp_window(window, text="C", x=5, y=73,
                                action=["bst-button-pressed",c,"under"],
                                height=24, width=40)
    actr.add_button_to_exp_window(window, text="Reset", x=5, y=123,
                                action="bst-reset-button-pressed",
                                height=24, width=65)

    actr.add_line_to_exp_window(window,[75,35],[a + 75,35],"black")
    actr.add_line_to_exp_window(window,[75,60],[b + 75,60],"black")
    actr.add_line_to_exp_window(window,[75,85],[c + 75,85],"black")
    actr.add_line_to_exp_window(window,[75,110],[goal + 75,110],"green")
```

The button_pressed function is associated with the A, B, and C buttons created above. It will be called when those buttons are pressed and passed the length of the corresponding stick and whether that button represents the over- or under- shoot strategy if it is the first one chosen.

```
def button_pressed(len,dir):
    global choice,current_stick
```

If a previous strategy choice hasn't been set, then set it to the current strategy.

```
    if not(choice):
        choice = dir
```

If the trial isn't over yet then update the current stick using the chosen one.

```
    if not(done):
        if current_stick > target:
            current_stick -= len
        else:
            current_stick += len

        update_current_line()
```

The reset_display function gets called when the reset button is pressed and if the trial isn't over yet it will remove the current stick so the participant can start over.

```
def reset_display():
    global current_stick

    if not(done):
        current_stick = 0
        update_current_line()
```

Here we add the commands for the previous two functions so that they can be called as button actions.

```
actr.add_command("bst-button-pressed",button_pressed,
                 "Choice button action for the Building Sticks Task.  Do not call directly")
actr.add_command("bst-reset-button-pressed",reset_display,
                 "Reset button action for the Building Sticks Task.  Do not call directly")
```

Update_current_line redraws the current line and updates the global variables which hold the current state as necessary.

```
def update_current_line():
    global current_line,done
```

If the current stick has the target length the task is over. Set the done flag, modify the line to show its new length, and display the done prompt.

```
if current_stick == target:
    done = True
    actr.modify_line_for_exp_window(current_line,[75,135],[target + 75,135])
    actr.add_text_to_exp_window(window,"Done", x=180, y=200)
```

If the current stick is returned to length 0 then remove it from the screen and clear the variable holding the line.

```
elif current_stick == 0:
    if current_line:
        actr.remove_items_from_exp_window(window,current_line)
        current_line = None
```

If there is a current stick then update it to the new length.

```
elif current_line:
    actr.modify_line_for_exp_window(current_line,[75,135],[current_stick + 75,135])
```

Otherwise, draw a new current stick of the appropriate length and save that item.

```
else:
    current_line = actr.add_line_to_exp_window(window,[75,135],
                                               [current_stick + 75,135],"blue")
```

The do_experiment function takes a list of lengths which represent the sticks to present on a trial and an optional parameter indicating whether a person will be performing the task. It calls build_display to create the window and then either waits for a person to finish the task or runs the model for up to 60 seconds to perform the task (in real time if the window is visible for the model). When running the model, since this task requires using the mouse it first uses start_hand_at_mouse to put the model's hand on the virtual mouse instead of the virtual keyboard because the keyboard is the default location for the hands when using the experiment window device.

```
def do_experiment(sticks, human=False):
    build_display(*sticks)

    if human:
        if actr.visible_virtuals_available():
            wait_for_human()
    else:
        actr.install_device(window)
        actr.start_hand_at_mouse()
        actr.run(60,visible)
```

The wait_for_human function loops until the task is over and then waits one second more after displaying the done prompt.

```
def wait_for_human ():
    while not(done):
        actr.process_events()

    start = actr.get_time(False)
    while (actr.get_time(False) - start) < 1000:
        actr.process_events()
```

The bst_set function takes three required parameters which indicate whether a person is doing the task, whether the window should be visible, and a list of stick length lists for the trials to present. It also takes an optional parameter which indicates whether the model should be learning from trial to trial or not. It loops over the trials provided, resetting the model if it isn't supposed to learn from trial to trial, and records the initial strategy choice which was made on each one. It returns the list of strategy choices.

```
def bst_set(human,vis,stims,learn=True):
    global visible

    result = []

    visible = vis

    for stim in stims:
        if not(learn) and not(human):
            actr.reset()
        do_experiment(stim,human)
        result.append(choice)

    return result
```

The test function takes one required parameter which is the number of two-trial test runs to perform and an optional parameter indicating whether a person will be performing the task. It runs the required number of trials and counts the number of times over-shoot is used to solve them returning the list of those counts.

```
def test(n,human=False):

    l = len(no_learn_stims)

    result = [0]*l

    if human or (n == 1):
        v = True
    else:
        v = False

    for i in range(n):

        d = bst_set(human,v,no_learn_stims,False)

        for j in range(l):
            if d[j] == "over":
                result[j] += 1

    return result
```

The experiment function takes one required parameter which is the number of times to perform the whole experiment and an optional parameter indicating whether a person will be performing the task. It runs the required number of experiments and computes the average proportion of times over-shoot is chosen for each trial and the average utility value of the critical productions in the model after each experiment. Those results are printed along with the correlation and mean-deviation from the original experimental data.

```
def experiment(n,human=False):
```

```
    l = len(exp_stims)

    result = [0] * l
    p_values = [["decide-over",0],["decide-under",0],["force-over",0],["force-under",0]]
    for i in range(n):
        actr.reset()
        d = bst_set(human,human,exp_stims)
        for j in range(l):
            if d[j] == "over":
                result[j] += 1

        actr.hide_output()

        for p in p_values:
            p[1]+=production_u_value(p[0])

        actr.unhide_output()

    result = list(map(lambda x: 100 * x / n,result))

    if len(result) == len(exp_data):
        actr.correlation(result,exp_data)
        actr.mean_deviation(result,exp_data)

    print()
    print("Trial ",end="")
    for i in range(l):
        print("%-8d"%(i + 1),end="")
    print()

    print("  ",end="")
    for i in range(l):
        print("%8.2f"%result[i],end="")

    print()
    print()

    for p in p_values:
        print("%-12s: %6.4f"%(p[0],p[1]/n))
```

The production_u_value function takes one parameter which should be the name of a production. It returns the current value of the :u parameter for that production.

```
def production_u_value(prod):
    return actr.spp(prod,":u")[0][0]
```

New Commands

Creating Buttons

add-button-to-exp-window and **add_button_to_exp_window –** these functions are similar to the add-text-to-exp-window function that you have seen many times before, but for creating a button object which can be pressed by a person or the model. It has one required parameter which is the window in which to display the button, and several keyword parameters for specifying the text to display on the button, the x and y pixel coordinates of the upper-left corner

of the button, an action to perform when the button is pressed, the height and width of the button in pixels, and a color for the background of the button. The action can be a string which names a valid command in ACT-R, a list with the name of a command and the parameters to pass to that command, or nil (Lisp)/None(Python). If no command is provided then pressing the button will result in a warning being printed when it is pressed to indicate the time that the press occurred. It returns an identifier for the button item.

Removing items

remove-items-from-exp-window and **remove_items_from_exp_window –** these functions take one required parameter and any number of additional parameters. The required parameter is an identifier for an experiment window. The remaining values should be the identifiers returned from items that were added to that window. Those items are then removed from that window.

Modifying lines

modify-line-for-exp-window and **modify_line_for_exp_window –** these functions take three required parameters. The first is the identifier for a line item that has been created. The next two are the lists with the x,y coordinates for the end points of the line. It has one optional parameter which should be the name of a color. The line object which is provided is updated so that its end points and color have the new values provided instead of the values they had previously. If a value of nil (Lisp) or None (Python) is provided for an end point then that item is not changed and remains as it was.

Using the mouse

By default, the experiment window device provides a keyboard and mouse for the model and its hands start on the keyboard. If you want the model's right hand to start on the mouse you can use the **start-hand-at-mouse** or **start_hand_at_mouse** function before you run the model. It is also possible for the model to move its hand explicitly to the mouse with a manual request using the cmd value hand-to-mouse and then to move it back to the keyboard with the cmd value hand-to-home.

Hiding ACT-R output

As you have seen throughout the tutorial, many of the ACT-R commands often print out information when they are used. Sometimes that output is not necessary, like in this task where we want to get the value of productions' utilities but don't care about seeing them each time. To avoid that unnecessary output there are some ways to turn it off or hide it, but there are some things to be careful about when doing so.

One option is to set the parameter :cmdt to the value nil. That parameter is similar to the :v parameter but it controls the output of the commands (it's the command trace parameter). The downside of turning that off is that it turns off all of the command output for the model regardless of the source of the command, which isn't really a concern when working through the tasks in the tutorial where you are the only one interacting with the system, but in the more general case that could be an issue if there are multiple tasks/interfaces connected some of which need to see command output and others which do not.

The best option is currently only available for code called from the ACT-R prompt i.e. Lisp. That is the **no-output** macro. It can be wrapped around any number of calls to ACT-R code and will suppress the output of those calls without affecting any commands being evaluated elsewhere e.g. from some other connected client. That is possible because of how macros work in Lisp and because it will be executing all of the ACT-R commands within a single thread thus it is able to make the change without interfering with other commands. That sort of temporary and local disabling is not possible through the remote interface because of how the remote commands are processed with respect to the current output mechanisms, but it is something which is being investigated for improvement in future versions.

When using the Python interface provided with the tutorial there are two pairs of functions that can be used to suppress and restore the output in the Python interface (note however that any other connected client will still get the output). The pair used in this task is **hide_output** and **unhide_output**. Those will suppress and then restore the printing of all the output generated by ACT-R in the Python interface – regardless of where that output is generated i.e. they will also disable warning messages which may be generated from elsewhere, for example those generated by a module while the model is running. These functions don't stop the Python interface from monitoring for output, just whether or not it is printed when it is received, and are recommended when one just wants to disable the output briefly and then enable it again. The other pair of functions are **stop_output** and **resume_output**. Those command stop monitoring for ACT-R output entirely and then restore the monitoring of output respectively. Those are costly operations and thus not the sort of thing one would want to do repeatedly, but could be useful when running a long task for which no output is needed where they will only be called once at the start and end of that task.

Providing rewards

Although it wasn't used in the task for this unit the main text mentioned the trigger-reward command. If one wants to provide a reward to a model for utility learning purposes independently of the productions then the **trigger-reward** (Lisp) or **trigger_reward** (Python) function can be used to do so. It takes one required parameter which is the amount of reward to provide and that reward will be given to the model at the current time. The reward can be a number, in which case the utilities are updated as described in the main unit text, or any other value which is not "null" (nil in Lisp or False/None in Python). If a non-numeric reward is provided that value is displayed in the model trace and serves to mark the last point of reward, but does not cause any updating of the utilities – it only serves to set the stopping point for how far back the next numeric reward will be applied.

Using values returned from calling ACT-R commands in productions

In the previous unit the !eval! production operator was described to show how one can call commands from within productions. It is also possible to use the value returned from such a call within the production. To do that the !bind! operator is used. It works very similarly to the !eval! operator, but it also requires specifying a variable name in which to store the result of evaluating the command. The encode-over and encode-under productions in the Building Sticks model use a !bind! to compute the difference between stick lengths and store the result in a slot. Here is the encode-under production:

```
(p encode-under
   =goal>
      isa         try-strategy
      state       encode-under
   =imaginal>
      isa         encoding
      b-loc       =b
      length      =goal-len
   =visual>
      isa         line
      width       =c-len
   ?visual>
      state       free
  ==>
   !bind! =val    (- =goal-len =c-len)

   =imaginal>
      under       =val
   =goal>
      state       encode-over
   +visual>
      isa         move-attention
      screen-pos  =b)
```

That line of the production is using a Lisp expression to be evaluated, but like !eval! it can also evaluate an ACT-R command given the string of its name. Here is an example which would store the result of evaluating "test-command" passed the value of the =number variable into the =result variable:

```
   !bind! =result ("test-command" =number)
```

On the LHS of a production a !bind! specifies a condition that must be met before the production can be selected just like all the other items on the LHS. The value returned by the evaluation of a LHS !bind! must be true for the production to be selected (where true is technically anything that is not nil in Lisp and if the command is implemented in some other language the values must not be the equivalent of nil e.g. False and None in Python are equivalent to nil in Lisp).

On the RHS of a production if the evaluated expression from a !bind! returns a nil result it will generate a warning from ACT-R and a value of t will be used instead since a variable in a production cannot be bound to nil since that indicates the absence of a value.

Like !eval!, !bind! can be a powerful tool which can easily be abused. It is not required, and should not be used, when writing any of the productions for the assignments in the tutorial (although in unit 7 some of the starting productions for the assignment do use a !bind!).