

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 is run as an individual trial which implemented 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. It could have been entirely event-driven, but for example purposes this shows that it's possible to combine those approaches.

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, and they create a button object which can be pressed by a person (if the window is visible) 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/None. If no action 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, but they can be added back if desired using the **add-items-to-exp-window** or **add_items_to_exp_window** commands (which are not used in the tutorial, but work like remove in that they require a window and then any number of items to add).

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 for a complicated task and some of those pieces need to see command output and others 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 commands 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 like 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 the !bind! operator 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)
```

In this case the `!bind!` operator 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!` to keep things simpler for the task).