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.

This document will briefly describe how the experiment for the Building Sticks Task (BST) differs from others you have seen, and then describe the new commands that it uses. After that, it will provide some information about a few other mechanisms in the software that may be useful when creating models.

BST implementation

In the 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 is 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, it was written with both styles to show that it is possible to combine those approaches when convenient.

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 of 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).

Additional Chunk-type Capabilities

In the main unit text it mentioned the ability of a chunk-type to specify default slot values and that one can create hierarchies of chunk-types. We will describe how that is done and how it affects the model in the following sections.

Default chunk-type slot values

If we look at the actions of the productions encode-line-goal and click-mouse from the building sticks model we see that do not specify the cmd slot in the requests to the **visual** and **manual** buffers which we have seen previously, and instead are only declaring a chunk-type with an isa:

```
(p encode-line-goal
   =goal>
     isa         try-strategy
     state       attending
   =imaginal>
     isa         encoding
     c-loc       =c
     goal-loc    nil
   =visual>
     isa         line
     screen-pos =pos
     width       =length
   ?visual>
```

```
        state       free
   ==>
    =imaginal>
       goal-loc    =pos
       length      =length
    =goal>
       state       encode-under
    +visual>
       isa         move-attention
       screen-pos =c)

(p click-mouse
   =goal>
       isa    try-strategy
       state  move-mouse
   ?manual>
       state  free
   ==>
    =goal>
       state  wait-for-click
    +manual>
       isa    click-mouse)
```

Up to this point it has been stated that the isa declarations are optional and not a part of a request or condition. If that is true then how are those requests doing the right thing since we've also said that **visual** and **manual** requests require the cmd slot be specified to indicate the action to perform?

The answer to that question is that there is an additional process which happens when declaring a chunk-type with isa both in creating chunks and in specifying the conditions and actions of a production. That additional process relies on the ability to indicate default values for a slot when creating a chunk-type. Up until now when we have created chunk-types we have only specified the set of slots which it can include, but in addition to that one can specify default initial values for specific slots. If a chunk-type indicates that a slot should have a value by default, then when that chunk-type is declared any slots with default values that are not specified in the chunk definition or production statement will automatically be included with their default values.

To specify a default value for a slot in a chunk-type one needs to specify a list of two items for the slot where the first item is the slot name and the second is the default value, instead of just a slot name. Here are some example chunk-types for reference:

```
  (chunk-type a slot)
  (chunk-type b (slot 1))
  (chunk-type c slot (slot2 t))
```

The chunk-type b has a default value of 1 for the slot named slot and chunk-type c has a default value of t for the slot named slot2. If we create some chunks specifying those chunk-types, some of which include a value for the slots with default values and others which do not, we can see how the default values are filled in for the chunks that do not specify those slots:

```
? (define-chunks
   (c1 isa a)
   (c2 isa b)
   (c3 isa b slot 3)
   (c4 isa c)
   (c5 isa c slot2 nil))
(C1 C2 C3 C4 C5)

? (pprint-chunks c1 c2 c3 c4 c5)
C1

C2
   SLOT  1

C3
   SLOT  3

C4
   SLOT2  T

C5

(C1 C2 C3 C4 C5)
```

For the slots with default values the chunks which specify the slot get the value specified, whereas the ones that do not get the default value. The same process applies to conditions and actions in a production. The chunk-types move-attention and click-mouse are created by the vision and motor modules and include default values for the cmd slot essentially like this (there is more to the real chunk-type specifications which will be discussed in the next section):

```
(chunk-type move-attention (cmd move-attention) screen-pos)
(chunk-type click-mouse (cmd click-mouse))
```

Therefore, this request:

```
   +visual>
      isa        move-attention
      screen-pos =c
```

is equivalent to this:

```
   +visual>
      cmd        move-attention
      screen-pos =c
```

We can also see that by looking at the actual representation of the production which the procedural module has for encode-line-goal using the Procedural viewer in the ACT-R Environment or by using the pp (print production) command:

```
? (pp encode-line-goal)
(P ENCODE-LINE-GOAL
```

```
    =GOAL>
        STATE ATTENDING
    =IMAGINAL>
        C-LOC =C
        GOAL-LOC NIL
    =VISUAL>
        SCREEN-POS =POS
        WIDTH =LENGTH
        LINE T
    ?VISUAL>
        STATE FREE
 ==>
    =IMAGINAL>
        GOAL-LOC =POS
        LENGTH =LENGTH
    =GOAL>
        STATE ENCODE-UNDER
    +VISUAL>
        SCREEN-POS =C
        CMD MOVE-ATTENTION
)
```

Notice how none of the isa declarations from the original definition are part of the actual production representation but the cmd slot has been included in the request to the **visual** buffer. If you inspect the click-mouse production you will find a similar result of a default value being inserted.

Chunk-type hierarchy

Now we will look at some of the isa declarations on the LHS of the encode-line-goal and read-done productions from the building sticks model.

```
(p encode-line-goal
   =goal>
      isa        try-strategy
      state      attending
   =imaginal>
      isa        encoding
      c-loc      =c
      goal-loc   nil
   =visual>
      isa        line
      screen-pos =pos
      width      =length
   ?visual>
      state      free
  ==>
   =imaginal>
      goal-loc   =pos
      length     =length
   =goal>
      state      encode-under
   +visual>
      isa        move-attention
      screen-pos =c)
```

```
(p read-done
   =goal>
      isa    try-strategy
      state  read-done
   =visual>
      isa    text
      value  "done"
  ==>
   +goal>
      isa    try-strategy
      state  start)
```

Previously in the tutorial it was stated that the chunks placed into the **visual** buffer are created with slots from a chunk-type named visual-object. However, if we look at the conditions in those productions they are specifying chunk-types of line and text respectively in the **visual** buffer conditions.

In addition to the chunk-type visual-object the vision module defines some other chunk-types for the objects which are used by the AGI experiment windows. Those other chunk-types, like line and text, are actually subtypes of the type visual-object. A subtype contains all of the slots that its parent chunk-type contains, and may also contain additional slots or different default slot values for the slots which it has. When a feature from an AGI window is converted into a chunk for the **visual** buffer it actually uses the slots of one of the specific types like line or text to create the chunk.

In general, one can create an arbitrary hierarchy of chunk-types with each subtype inheriting the slots and default values of its parent chunk-type (or even multiple parent chunk-types). A hierarchy of chunk-types can be helpful to the modeler in specifying chunks and productions, but other than through the automatic use of default slot values, such a hierarchy has no effect on the actual chunks or productions in the model.

To create a chunk-type which is a subtype of another chunk-type that parent type must be specified in the definition of the subtype. That is done by using a list to specify the name of the subtype which includes a list which has the keyword :include and the name of a parent chunk-type for each parent type to be included. These chunk-type definitions create chunk-types a and b, and then a chunk-type named c which is a subtype of chunk-type a and a chunk-type d which is a subtype of both a and b:

```
  (chunk-type a slot1)
  (chunk-type b slot2)
  (chunk-type (c (:include a)) slot3)
  (chunk-type (d (:include a) (:include b)) slot4)
```

One might think these definitions would be equivalent to those shown above:

```
(chunk-type a slot1)
(chunk-type b slot2)
(chunk-type c slot1 slot3)
(chunk-type d slot1 slot2 slot4)
```

However they are not equivalent because the subtyping mechanism is also extending the slots which are valid for the parent types as well. In addition to a subtype inheriting the slots from its parent type(s), the parent types also gain access to all of the slots from their children. Thus, with the definitions using the

hierarchy it is acceptable to specify a parent type and use slots defined in its subtypes, but that is not valid for chunks which just happen to have the same slot names. Thus, with the first set of chunk-type definitions this is acceptable:

```
(define-chunks (isa a slot3 t slot4 10))
```

because chunk-types c and d are subtypes of chunk-type a and have slots named slot3 and slot4, but with the second set that would result in warnings for invalid slots in the chunk definition.

These are the chunk-type definitions for the line and text chunk-types which are used to create the **visual** buffer chunks for the corresponding AGI items:

```
(chunk-type visual-object screen-pos value status color height width)
(chunk-type (text (:include visual-object)) (text t))
(chunk-type (line (:include visual-object)) (line t) end1-x end1-y end2-x end2-y)
```

The text and line chunk-types are subtypes of the visual-object type and each includes a new slot with the same name as the type and a default value of t. Because of that default slot value, declaring the **visual** buffer tests with the types text and line in these productions adds that additional condition to the productions as we can see when we look at the actual representation of them:

```
(P ENCODE-LINE-GOAL
   =GOAL>
        STATE ATTENDING
   =IMAGINAL>
        C-LOC =C
        GOAL-LOC NIL
   =VISUAL>
        SCREEN-POS =POS
        WIDTH =LENGTH
        LINE T
   ?VISUAL>
        STATE FREE
 ==>
   =IMAGINAL>
        GOAL-LOC =POS
        LENGTH =LENGTH
   =GOAL>
        STATE ENCODE-UNDER
   +VISUAL>
        SCREEN-POS =C
        CMD MOVE-ATTENTION
)
(P READ-DONE
   =GOAL>
        STATE READ-DONE
   =VISUAL>
        VALUE "done"
        TEXT T
 ==>
   +GOAL>
```

```
      STATE START
)
```

Those slot tests could have been specified directly, but declaring the type can be more readable and is consistent with the way chunk-types were used in older versions of ACT-R (those prior to version 6.1).

In the default slot section above it showed chunk-type definitions for the move-attention and click-mouse actions to indicate they had default slots, but the actual definitions of those chunk-types also include parent types of vision-command and motor-command respectively which could also be used (though they would require specifying the cmd slot's value).