Unit 2 Code Description

This document will describe the code that implements the experiments from unit 2, how ACT-R is interfaced to them, and some commands that can be used from the prompt to get information about the model instead of using the Environment tools. Before getting into the specifics of the code and commands however we will first describe a component of the ACT-R software which is used to create the experiments and provide a little more information on using the Python interface. Finally, there is also a section at the end of the document with some details about the underlying interface which enables the connection between the core ACT-R software and arbitrary tasks or tools (like the interactive Python interface and the ACT-R Environment).

ACT-R GUI Interface

All of the experiments which are built for the models in the tutorial will be created using a set of interface tools provided with the ACT-R software which we call the AGI (ACT-R GUI Interface). The AGI allows for the creation of simple tasks which can be composed of text, buttons, and lines and interacted with using the keyboard and mouse. When the ACT-R Environment is running, the AGI tasks can be displayed in real windows which can be interacted with by either a person or an ACT-R model (as we saw in the experiments of this unit). Whether the ACT-R Environment is running or not, the AGI can also create a virtual interface which does not display a real window but which can still be interacted with by an ACT-R model exactly the same way as it does with the real window – there is no difference between the real and virtual interface from a model's perspective. The advantage of using a virtual interface for the model is that it is much faster to run the task with a virtual interface than it is a real one, but the downside is that you cannot see the task to monitor what the model is doing which can be important while developing the model and working out any problems in its operation. That is why the AGI provides the model the exact same interface regardless of whether it is a real or virtual window – you can create the task using a real window while developing the model and then change it to a virtual window once the model is working correctly to be able to run it through the task faster for collecting data over multiple trials.

It is not required that one create tasks for an ACT-R model using the AGI. It is also possible to provide features directly to the vision module, and have the model use the virtual keyboard and mouse without an AGI window as well as create new motor interface devices. However, that level of interaction will not be shown in the tutorial.

One final note on the AGI is that it was designed for creating tasks for ACT-R models. The tasks it creates can be interacted with by real participants, but it was not designed with that use in mind. In particular, when running with a real participant it does not make any claims as to the accuracy of the timing information it can collect or the latency of the visual presentations and input responses. For interaction with the model those are not an issue since the model runs in a simulated time frame where the exact time is always available instantly and the clock can pause arbitrarily long before advancing to allow for instantaneous presentations and no-latency on input responses. Therefore, we do not

recommend using the AGI to create experiments for real participants if any timing information is to be collected.

Additional Python interface information

Before describing the details of both the Lisp and Python implementations of the experiments, we will provide a little more information on the Python interface to ACT-R. We will also show how you can use some of the ACT-R commands, which the unit 1 code document described being used at the ACT-R prompt, from the Python prompt as well.

The actual interface between Python and the ACT-R software is provided by another Python module called actr which is also located in the python directory of the software. That module provides the code that handles the remote interface described in the appendix of this text, and also defines Python functions which correspond to many of the ACT-R commands available through the remote interface to make them easier to use. That module gets imported by all of the modules for the experiments to enable the interface, and it could also be imported directly if you want easier access to the functions for interacting with ACT-R from the prompt. Once you have done that you can then use the available functions from that module. In general, the Python functions will have the same name as the corresponding command in ACT-R, but with all of the "-" characters replaced with "_" characters to make them valid Python function names. We will describe many of the available functions as we progress through the tutorial, and we will start here with the ones that correspond to the commands used at the ACT-R prompt that where shown in the unit 1 code document: **reset**, **reload**, **run**, **load-act-r-model**, **buffer-chunk**, **dm**, **sdm**, and **whynot**.

The first four of those work the same as the commands described for the ACT-R prompt, and here is an example showing the addition model from unit 1 being loaded, run for 1 second, reloaded, run for .1 seconds, and then being reset.

```
>>> import actr
ACT-R connection has been started with default parameters.
>>> actr.load_act_r_model("ACT-R:tutorial;unit1;addition.lisp")
True
>>> actr.run(1)
     0.000   GOAL                 SET-BUFFER-CHUNK GOAL SECOND-GOAL NIL
     0.000   PROCEDURAL           CONFLICT-RESOLUTION
     0.050   PROCEDURAL           PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL           CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE          start-retrieval
     0.050   PROCEDURAL           CONFLICT-RESOLUTION
     0.100   DECLARATIVE          RETRIEVED-CHUNK F
     0.100   DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL           CONFLICT-RESOLUTION
     0.150   PROCEDURAL           PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL           CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE          start-retrieval
     0.150   PROCEDURAL           CONFLICT-RESOLUTION
     0.200   DECLARATIVE          RETRIEVED-CHUNK A
     0.200   DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL A
```

```
     0.200   PROCEDURAL            CONFLICT-RESOLUTION
     0.250   PROCEDURAL            PRODUCTION-FIRED INCREMENT-COUNT
     0.250   PROCEDURAL            CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE           start-retrieval
     0.250   PROCEDURAL            CONFLICT-RESOLUTION
     0.300   DECLARATIVE           RETRIEVED-CHUNK G
     0.300   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL G
     0.300   PROCEDURAL            CONFLICT-RESOLUTION
     0.350   PROCEDURAL            PRODUCTION-FIRED INCREMENT-SUM
     0.350   PROCEDURAL            CLEAR-BUFFER RETRIEVAL
     0.350   DECLARATIVE           start-retrieval
     0.350   PROCEDURAL            CONFLICT-RESOLUTION
     0.400   DECLARATIVE           RETRIEVED-CHUNK B
     0.400   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL B
     0.400   PROCEDURAL            CONFLICT-RESOLUTION
     0.450   PROCEDURAL            PRODUCTION-FIRED INCREMENT-COUNT
     0.450   PROCEDURAL            CLEAR-BUFFER RETRIEVAL
     0.450   DECLARATIVE           start-retrieval
     0.450   PROCEDURAL            CONFLICT-RESOLUTION
     0.500   DECLARATIVE           RETRIEVED-CHUNK H
     0.500   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL H
     0.500   PROCEDURAL            PRODUCTION-FIRED TERMINATE-ADDITION
7
     0.500   PROCEDURAL            CONFLICT-RESOLUTION
     0.500   ------                Stopped because no events left to process
[0.5, 74, None]
>>> actr.reload()
True
>>> actr.run(.1)
     0.000   GOAL                  SET-BUFFER-CHUNK GOAL SECOND-GOAL
NIL
     0.000   PROCEDURAL            CONFLICT-RESOLUTION
     0.050   PROCEDURAL            PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL            CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE           start-retrieval
     0.050   PROCEDURAL            CONFLICT-RESOLUTION
     0.100   DECLARATIVE           RETRIEVED-CHUNK F
     0.100   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL            CONFLICT-RESOLUTION
     0.100   ------                Stopped because time limit reached
[0.1, 20, None]
>>> actr.reset()
True
```

The other ACT-R commands from unit 1, **dm**, **sdm**, **buffer-chunk**, and **whynot**, require a slightly different syntax when called from Python compared to the version called from the ACT-R prompt. In the ACT-R version we could just specify the arguments for the commands without any additional syntactic markers, for example, here is the **buffer-chunk** command being used to get the chunk from the goal buffer at this time:

```
? (buffer-chunk goal)
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
```

```
   ARG2  2
   SUM  6
   COUNT  1

(SECOND-GOAL-0)
```

From Python however we must specify the arguments as strings. Here we will show those being used, continuing from where we left off in the example above. First, we will run the model for .3 seconds so that there are some chunks in the buffers to view:

```
>>> actr.run(.3)
     0.000   GOAL                    SET-BUFFER-CHUNK GOAL SECOND-GOAL NIL
     0.000   PROCEDURAL              CONFLICT-RESOLUTION
     0.050   PROCEDURAL              PRODUCTION-FIRED INITIALIZE-ADDITION
     0.050   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.050   DECLARATIVE             start-retrieval
     0.050   PROCEDURAL              CONFLICT-RESOLUTION
     0.100   DECLARATIVE             RETRIEVED-CHUNK F
     0.100   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL F
     0.100   PROCEDURAL              CONFLICT-RESOLUTION
     0.150   PROCEDURAL              PRODUCTION-FIRED INCREMENT-SUM
     0.150   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.150   DECLARATIVE             start-retrieval
     0.150   PROCEDURAL              CONFLICT-RESOLUTION
     0.200   DECLARATIVE             RETRIEVED-CHUNK A
     0.200   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL A
     0.200   PROCEDURAL              CONFLICT-RESOLUTION
     0.250   PROCEDURAL              PRODUCTION-FIRED INCREMENT-COUNT
     0.250   PROCEDURAL              CLEAR-BUFFER RETRIEVAL
     0.250   DECLARATIVE             start-retrieval
     0.250   PROCEDURAL              CONFLICT-RESOLUTION
     0.300   DECLARATIVE             RETRIEVED-CHUNK G
     0.300   DECLARATIVE             SET-BUFFER-CHUNK RETRIEVAL G
     0.300   PROCEDURAL              CONFLICT-RESOLUTION
     0.300   ------                  Stopped because time limit reached
[0.3, 46, None]
```

Here is the **buffer_chunk** function being called to print the chunk from the goal buffer:

```
>>> actr.buffer_chunk('goal')
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1
['SECOND-GOAL-0']
```

The return value is a list of the names of the chunks in the requested buffers (also represented as strings). That can be called with any number of buffers (including 0). Here is what happens if we request the retrieval and visual buffers now:

```
>>> actr.buffer_chunk('retrieval','visual')
RETRIEVAL: G-0 [G]
```

```
G-0
   FIRST  6
   SECOND  7
VISUAL: NIL
['G-0', None]
>>>
```

The return value for the visual buffer is None because the buffer is empty. If we call it with no buffers it prints out the contents of all buffers and returns a list of lists where each sublist contains the name of a buffer as the first element, and if the buffer contains a chunk the second element is the name of that chunk:

```
>>> actr.buffer_chunk()
RETRIEVAL: G-0 [G]
IMAGINAL: NIL
MANUAL: NIL
GOAL: SECOND-GOAL-0
IMAGINAL-ACTION: NIL
VOCAL: NIL
AURAL: NIL
PRODUCTION: NIL
VISUAL-LOCATION: NIL
AURAL-LOCATION: NIL
TEMPORAL: NIL
VISUAL: NIL
[['RETRIEVAL', 'G-0'], ['IMAGINAL'], ['MANUAL'], ['GOAL', 'SECOND-GOAL-
0'], ['IMAGINAL-ACTION'], ['VOCAL'], ['AURAL'], ['PRODUCTION'], ['VISUAL-
LOCATION'], ['AURAL-LOCATION'], ['TEMPORAL'], ['VISUAL']]
>>>
```

The **dm** function can be used to print all of the chunks in declarative memory and return a list of their names, or to print only those chunks specified (again using strings to provide the names):

```
>>> actr.dm()
SECOND-GOAL
   ARG1  5
   ARG2  2
J
   FIRST  9
   SECOND  10
I
   FIRST  8
   SECOND  9
H
   FIRST  7
   SECOND  8
G
   FIRST  6
   SECOND  7
F
   FIRST  5
   SECOND  6
E
   FIRST  4
   SECOND  5
```

```
D
   FIRST  3
   SECOND  4
C
   FIRST  2
   SECOND  3
B
   FIRST  1
   SECOND  2
A
   FIRST  0
   SECOND  1
['SECOND-GOAL', 'J', 'I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']

>>> actr.dm('a','b','c')
A
   FIRST  0
   SECOND  1
B
   FIRST  1
   SECOND  2
C
   FIRST  2
   SECOND  3
['A', 'B', 'C']
```

For the **sdm** function to search declarative memory we again need to specify the constraints using strings. Here is a search for all of the items which do not have the value of 1 in their first slot:

```
>>> actr.sdm('-','first','1')
SECOND-GOAL
   ARG1  5
   ARG2  2
A
   FIRST  0
   SECOND  1
C
   FIRST  2
   SECOND  3
D
   FIRST  3
   SECOND  4
E
   FIRST  4
   SECOND  5
F
   FIRST  5
   SECOND  6
G
   FIRST  6
   SECOND  7
H
   FIRST  7
   SECOND  8
I
```

```
   FIRST  8
   SECOND  9
J
   FIRST  9
   SECOND  10
['SECOND-GOAL', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

Because the value for the slot is an integer it could actually be specified without encoding it into a string and get the same result:

```
>>> actr.sdm('-','first',1)
SECOND-GOAL
   ARG1  5
   ARG2  2
A
   FIRST  0
   SECOND  1
C
   FIRST  2
   SECOND  3
D
   FIRST  3
   SECOND  4
E
   FIRST  4
   SECOND  5
F
   FIRST  5
   SECOND  6
G
   FIRST  6
   SECOND  7
H
   FIRST  7
   SECOND  8
I
   FIRST  8
   SECOND  9
J
   FIRST  9
   SECOND  10
['SECOND-GOAL', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

The **whynot** function also requires that you specify the production names to test using strings and it returns a list of strings which name the productions which do match the current state (regardless of whether they were in the list being tested):

```
>>> actr.whynot('initialize-addition','increment-count')

Production INITIALIZE-ADDITION does NOT match.
(P INITIALIZE-ADDITION
   =GOAL>
       ARG1 =NUM1
       ARG2 =NUM2
```

```
       SUM NIL
 ==>
   =GOAL>
       SUM =NUM1
       COUNT 0
   +RETRIEVAL>
       FIRST =NUM1
)
It fails because:
The chunk in the GOAL buffer has the slot SUM.

Production INCREMENT-COUNT does NOT match.
(P INCREMENT-COUNT
   =GOAL>
       SUM =SUM
       COUNT =COUNT
   =RETRIEVAL>
       FIRST =COUNT
       SECOND =NEWCOUNT
 ==>
   =GOAL>
       COUNT =NEWCOUNT
   +RETRIEVAL>
       FIRST =SUM
)
It fails because:
The value in the FIRST slot of the chunk in the RETRIEVAL buffer does
not satisfy the constraints.
['INCREMENT-SUM']
>>>
```

One thing which you may have noticed is that when you call those functions from Python the output from ACT-R is shown in both the ACT-R window and in your Python session. The same is true if you call the corresponding function from the ACT-R prompt – both interfaces will display the output regardless of how it was generated. If you find that distracting or confusing when working from the Python prompt you can disable the output in the ACT-R window by calling the turn-off-act-r-output command at the ACT-R prompt:

```
? (turn-off-act-r-output)
```

Now that we have shown how ACT-R commands can be used from Python we will describe the Lisp and Python programs which implement the experiments from this unit.

Experiment Code

Below we will show the code which implements the experiments from the unit (in both Lisp and Python) and describe how it works. Before that however we will describe some general details about the structure of the experiment code which has been written for all of the tutorial tasks.

When writing these experiments we have tried to keep the implementations of the tasks fairly straight forward to make it easy to follow how they work. We have also tried to keep the two implementations as similar as possible for comparison purposes. That might not always lead to the most efficient or best looking code, but should help to facilitate the objective of this tutorial – to demonstrate how to use the ACT-R software for creating models and experiments for those models. For many of the experiments in the tutorial there will typically be one function that runs the experiment for either a model or a person, and that function will take an optional parameter which if specified as true (**t** in Lisp and **True** in Python) will run a person instead of the model. Most of the code to perform the task will be 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. The code could have been written using separate functions for a model and a human participant, but by using one function it is easier to see where the differences are.

Now we will look at the code for the experiments and provide some description of what it is doing, and also highlight the new ACT-R and AGI functions used in this unit. Those new functions will be described in greater detail at the end of this text. The actual code will be displayed in the same font that has been used for the examples and the descriptions will be in the same font as this paragraph.

Demo2 Lisp

The first thing that the code does is load the corresponding model which can perform the task.

```
(load-act-r-model "ACT-R:tutorial;unit2;demo2-model.lisp")
```

It creates a global variable which will be set to the string naming the key which is pressed.

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

A function is defined which will be called with two parameters. The first of those parameters is not used, but the second one will be the name of a key which is pressed. That key is stored in the *response* variable and then it clears the experiment window using the AGI function clear-exp-window.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))

  (setf *response* key)
  (clear-exp-window))
```

Here is the demo2-experiment function which runs the task. It takes one optional parameter which can be specified as t (or actually any non-nil value) to indicate a person is doing the task.

```
(defun demo2-experiment (&optional human)
```

It starts by resetting the system using the reset function described in unit 1 to make sure that the model is restored to its starting state.

```
  (reset)
```

Then it creates three local variables. Items is a randomly permuted list of the possible letters to display. Text1 is set to the first element from the items list, and window is set to the result of creating a new window for the task titled “Letter recognition” using the open-exp-window function.

```
  (let* ((items (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 items))
         (window (open-exp-window "Letter recognition")))
```

The letter from text1 is added to that window using the add-text-to-exp-window function.

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

The next two function calls are how we get the respond-to-key-press function defined above to be called when a key is pressed in the experiment. First, it uses the add-act-r-command function to create a new command in ACT-R named demo2-key-press which is associated with our respond-to-key-press function and it provides a documentation string to go along with that new command. That new command “demo2-key-press” is then set to monitor the output-key command using the monitor-act-r-command function. A command that monitors another command in this way will be called after that other command and it will be provided the same parameters that the monitored command was. In this case we are monitoring the output-key command because that is how the keyboard associated with experiment windows of the AGI indicate that a key was pressed.

```
    (add-act-r-command "demo2-key-press" 'respond-to-key-press
                       "Demo2 task output-key monitor")
    (monitor-act-r-command "output-key" "demo2-key-press")
```

The global variable *response* is set to nil to indicate that no key has been pressed.

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

Here we check the value of human to determine how to run the experiment.

```
    (if human
```

If it is a human doing the task then we loop using the while function until the *response* variable is set to a non-nil value calling the process-events command of the AGI repeatedly to allow the system to respond to external input.

```
(while (null *response*)
  (process-events))
```

If the model is doing the task (not a human) then we need to first tell the model what interface it is interacting with, which in this case is the experiment window which we have created, using the install-device function. Then we run the model for up to 10 seconds. You will also notice that in addition to specifying the time we have also passed the value t to the run command. The second parameter to run is optional, but if a true value is given then the model is run in "real time" which means that its processing of events is synchronized with the actual passing of time instead of running with a simulated clock that goes as fast as possible. When running models with a visible window it is often helpful to run them in real time so that you can actually see what they are doing.

```
(progn
  (install-device window)
  (run 10 t)))
```

Now that the experiment is done we stop monitoring the output-key command with our demo2-key-press command using the remove-act-r-command-monitor function, and then remove our demo2-key-press command from the system using the remove-act-r-command function. These two steps are done as cleanup for safety reasons because we do not want our function to be called when keys are output in any other experiment windows in the future (for example if you load the assignment experiment and try to run it), and to be extra safe we remove the command that is tied to our respond-to-key-press function entirely.

```
(remove-act-r-command-monitor "output-key" "demo2-key-press")
(remove-act-r-command "demo2-key-press")
```

The global variable *response* is then returned from the demo2-experiment function.

```
*response*))
```

Demo2 Python

The first thing that this code does is import the actr module to provide the interface to ACT-R.

```
import actr
```

Then it loads the corresponding model which is able to perform this task.

```
actr.load_act_r_model("ACT-R:tutorial;unit2;demo2-model.lisp")
```

It creates a global variable which will be set to the string naming the key which is pressed.

```
response = False
```

A function is defined which will be called with two parameters. The first of those parameters is not used, but the second one will be the name of a key which is pressed. That key is stored in the global response variable and then it clears the experiment window using the AGI function clear_exp_window.

```
def respond_to_key_press (model,key):
    global response

    response = key
    actr.clear_exp_window()
```

Here is the experiment function which runs the task. It takes one optional parameter which can be specified as True to indicate a person is doing the task.

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

It starts by resetting the system using the reset function described in unit 1 to make sure that the model is restored to its starting state.

```
    actr.reset()
```

Then it creates three local variables. Items is a randomly permuted list of the possible letters to display. Text1 is set to the first element from the items list, and window is set to the result of creating a new window for the task titled “Letter recognition” using the open_exp_window function.

```
    items = actr.permute_list(["B","C","D","F","G","H","J","K","L",
                               "M","N","P","Q","R","S","T","V","W",
                               "X","Y","Z"])
    text1 = items[0]
    window = actr.open_exp_window("Letter recognition")
```

The letter from text1 is added to that window using the add_text_to_exp_window function.

```
actr.add_text_to_exp_window(window, text1, x=125, y=150)
```

The next two function calls are how we get the respond_to_key_press function defined above to be called when a key is pressed in the experiment. First, it uses the add_command function to create a new command in ACT-R named demo2-key-press which is associated with our respond_to_key_press function and it provides a documentation string to go along with that new command. That new command demo2-key-press is then set to monitor the output-key command using the monitor_command function. A command that monitors another command in this way will be called after that other command and it will be provided the same parameters that the monitored command was. In this case we are monitoring the output-key command because that is how the keyboard associated with experiment windows of the AGI indicate that a key was pressed.

```
actr.add_command("demo2-key-press",respond_to_key_press,
                 "Demo2 task output-key monitor")
actr.monitor_command("output-key","demo2-key-press")
```

The global variable response is set to False to indicate that no key has been pressed.

```
global response
response = False
```

Here we check the value of human to determine how to run the experiment.

```
if human == True:
```

If it is a human doing the task then we loop until the response variable is not False calling the process_events function of ACT-R repeatedly to allow the system to respond to external input.

```
    while response == False:
        actr.process_events()
```

If it is not a human doing the task then perform the steps needed to run the model.

```
else:
```

Tell the model what interface it is interacting with, which in this case is the experiment window which we have created, using the install_device function. Then we run the model for up to 10 seconds. You will also notice that in addition to specifying the time we have also passed the value True to the run function. The second parameter to run is optional, but if a True value is given then the model is run in “real time” which means that its processing of events is synchronized with the actual passing of time instead of running with a simulated clock that goes as fast as possible. When running models with a

visible window it is often helpful to run them in real time so that you can actually see what they are doing.

```
actr.install_device(window)
actr.run(10,True)
```

Now that the experiment is done we stop monitoring the output-key command with our demo2-key-press command using the remove_command_monitor function, and then remove our demo2-key-press command from the system using the remove_command function. These two steps are done as cleanup for safety reasons because we do not want our function to be called when keys are output in any other experiment windows in the future (for example if you load the assignment experiment and try to run it), and to be extra safe we remove the command that is tied to our respond_to_key_press function entirely.

```
actr.remove_command_monitor("output-key","demo2-key-press")
actr.remove_command("demo2-key-press")
```

The global variable response is then returned from the experiment function.

```
return response
```

Unit2 code

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. That only involves the use of one ACT-R function which was not shown above, and only the sections of the unit2 functions which use that new function will be described here. Also, since both the Lisp and Python versions are so similar we will use the same description to cover both code segments.

After permuting the list of letters the target variable is set to the first letter and the foil variable is set to the second letter. Three variables called text1, text2, and text3 are initially set to the foil letter. A random integer from 0 to 2 is generated using a random function from ACT-R (either act-r-random in Lisp or random from the actr module in Python). That number is used to determine which of the three text variables should hold the target item, and then the three letters are added to the window.

Lisp

```
(let* ((items (permute-list '("B" "C" "D" "F" "G" "H" "J" "K" "L" "M" "N"
                              "P" "Q" "R" "S" "T" "V" "W" "X" "Y" "Z")))
       (target (first items))
       (foil (second items))
       (window (open-exp-window "Letter difference"))
       (text1 foil)
       (text2 foil)
```

```
      (text3 foil)
      (index (act-r-random 3)))

(case index
  (0 (setf text1 target))
  (1 (setf text2 target))
  (2 (setf text3 target)))

(add-text-to-exp-window window text1 :x 125 :y 75)
(add-text-to-exp-window window text2 :x 75 :y 175)
(add-text-to-exp-window window text3 :x 175 :y 175)
```

Python

```
items = actr.permute_list(["B","C","D","F","G","H","J","K","L",
                           "M","N","P","Q","R","S","T","V","W",
                           "X","Y","Z"])
target = items[0]
foil = items[1]
window = actr.open_exp_window("Letter difference")
text1 = foil
text2 = foil
text3 = foil
index = actr.random(3)

if index == 0:
   text1 = target
elif index == 1:
   text2 = target
else:
   text3 = target

actr.add_text_to_exp_window(window, text1, x=125, y=75)
actr.add_text_to_exp_window(window, text2, x=75, y=175)
actr.add_text_to_exp_window(window, text3, x=175, y=175)
```

The reason that we use ACT-R's random function instead of any native random function that may be available is because we can set the seed of the ACT-R random function using the :seed parameter in the sgp call of the model. That allows us to reproduce the same 'random' sequence with a model and task regardless of how it is being run which can be extremely useful when trying to debug a random problem with a model or when creating consistent example runs.

Safety note

Before describing the new ACT-R commands in detail there is one note about the code for these experiments which should be pointed out. The functions that are handling the key press are being called during the execution of other functions (process-events when a person does the task and run when the model does the task). That happens because the monitoring functions get called in a separate thread from the one which is running the

main task with the way the ACT-R interfaces have been created. Since both threads are accessing the same global variable (response) the safe thing to do would be to include the appropriate protection on that to avoid problems (a lock, semaphore, or some other construct available in the language used). However, since all we are looking for here is a simple change to the value and only one of the threads sets the variable there should not be any problems with the operation of this code and that protection has been ignored for the purpose of keeping the example task easy to read. If you are creating more complicated tasks which are using monitors for things like key presses and mouse clicks or other multi-threaded actions then you may need to put in the necessary protection for access to shared resources (like global variables) to avoid problems.

New ACT-R commands

Below we will provide more details on the new functions used in creating these tasks, and they will be grouped based on their general purpose. Many of these commands will be used in most of the tasks throughout the tutorial.

AGI commands

creating a window

The **open-exp-window** and **open_exp_window** functions are used to open a window to display a task which can be interacted with by either an ACT-R model or a real person. The function requires one parameter which is a string containing the title for that window, and that title should be unique i.e. only one window with a given title may be open at a time. If the name specified is the name of a window which was created previously then that existing window will be closed first, and then a new window created. The return value of that function is a window description which can be passed to other AGI functions for indicating which window to operate on and it is also a valid device list that can be installed for the model to interact with. There are also several other parameters which may be provided when creating a window and those will be described in later units when used by the tasks involved.

displaying text in a window

The **add-text-to-exp-window** and **add_text_to_exp_window** functions display text in a window that was opened using **open-exp-window** or **open_exp_window**. It has two required parameters. The first is a window description to indicate which window, and the second is a string of the text to display. It has multiple additional parameters which are accessed using keyword parameters in Lisp and keyword arguments in Python. Two of them are used in this unit's tasks: x and y. Those are the x and y coordinate within the window at which the upper-left corner of the text to be displayed will be positioned and should be integers (the upper-left corner of the window is 0,0 with x increasing to the right and y increasing toward the bottom). It returns a descriptor for the text item which

can be used to remove or change that item, but the details of that descriptor are not part of the specification and it should not be used for any other purpose.

clearing a window

The **clear-exp-window** and **clear_exp_window** functions are used to clear all items from a window which was opened using **open-exp-window** or **open_exp_window**. It has one optional parameter which if provided should be a window description. If only one window has been opened then the optional parameter is not needed and that open window will be the one cleared. It removes all of the items that have been added to that window.

key presses

When a key is pressed in a window which was created with **open-exp-window** or **open_exp_window** by a person or on the corresponding virtual keyboard device which is installed for models interacting with those windows, the ACT-R command **output-key** is called. That command is called with two parameters. The first is the name of a model which made the key press if it was made by a model or a value of **nil** (Lisp) or **None** (Python) if it was by a person interacting with the window. The second will be a string indicating the key which was pressed. When the model makes the action the event is also shown in the trace as seen in the last line of this segment of the trace from running the demo2 task:

```
0.485   PROCEDURAL        MODULE-REQUEST MANUAL
0.485   PROCEDURAL        CLEAR-BUFFER IMAGINAL
0.485   PROCEDURAL        CLEAR-BUFFER MANUAL
0.485   MOTOR             PRESS-KEY KEY v
0.485   PROCEDURAL        CONFLICT-RESOLUTION
0.735   MOTOR             PREPARATION-COMPLETE
0.735   PROCEDURAL        CONFLICT-RESOLUTION
0.785   MOTOR             INITIATION-COMPLETE
0.785   PROCEDURAL        CONFLICT-RESOLUTION
0.885   KEYBOARD          output-key DEMO2 v
```

To detect and record key presses from the experiment windows one will need to monitor the output-key command as described below.

Model interaction with tasks

To have a model interact with a task it must be told which ‘devices’ to use by using the **install-device** or **install_device** function. That function requires one parameter which must be a specification of a device for the model. A device is a general term for the types of things that a model can interact with and a window created by the **open-exp-window** or **open_exp_window** functions is a valid device. The device (or possibly multiple devices) which are installed for a model indicate where its percepts come from and/or where its output actions will go. The windows of the AGI provide visual percepts and they also install virtual keyboard and mouse devices with which the model can interact as

well as a virtual microphone for recording the model's speech. Creating new devices for a model is one way to interface a model to new environments, but that is beyond the scope of the tutorial.

Running a model

The **run** function was described in unit 1, but here we see that it takes an optional second parameter. The **run** function causes the simulated clock for the ACT-R system to advance which allows the model(s) to perform actions. The first parameter to **run** indicates the maximum amount of time that the system will be allowed to run specified in seconds. The second parameter is optional, but if provided as true (**t** in Lisp or **True** in Python) then the simulated clock for ACT-R will advance in step with the real passage of time instead of as fast as possible. The optional parameter can also be specified as a number to indicate a desired scaling of model time to real time (a value of 1 is the same as specifying true), but there are no guarantees on the ability to achieve a desired scaling – you can request a million to one scaling but it is unlikely to actually be able to achieve that.

Miscellaneous ACT-R functions

The **act-r-random** or **random** function can be used to return a pseudo-random number based on an initial seed which can be specified using the **sgp** command in a model. It requires one parameter which must be a number. If the number provided is an integer then the return value will be an integer chosen uniformly from 0 to that number minus 1. If it is a non-integer real number, N, then a real number uniformly chosen from the range of [0,N) will be returned. Using the random function provided by ACT-R when creating tasks allows one to generate the same "random" sequence of events for a model and task regardless of how the task is being run by specifying the same initial seed in the model definition. The specific algorithm used for the ACT-R random numbers is currently the MT19937 generator.

The **permute-list** or **permute_list** function can be used to permute the items in a list using the ACT-R random number generator to do so. They require one parameter which must be a list of items and it returns a randomly ordered copy of that list.

When a waiting loop is needed to collect a real response from an AGI window the **process-events** or **process_events** function should be called in that loop to allow the system a chance to handle the user interactions. It takes no parameters. It ensures that other threads are allowed to run which may be necessary for the system to be able to handle things like calling a monitoring function in response to a user pressing a key.

For the Lisp interface to ACT-R we have provided a **while** macro as 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. This is not really necessary because there are several other looping constructs available in Lisp, but a simple while can be easier to understand for novice Lisp programmers and it makes it easier to create a nearly line-to-line correspondence between the Lisp and Python versions of the tasks.

Like the buffer-chunk function which was shown in the previous unit to access the contents of a buffer, there are also **buffer-status** and **buffer_status** functions which will provide the status of the queryable information from the buffer (which is also shown in the "Buffers" tool of the Environment using the "Status" button described in this unit). It can be called with the names of any number of buffers and will print out the current status information for the queries of those buffers. Here are examples of calling it at the ACT-R prompt and in Python.

```
? (buffer-status goal retrieval)
GOAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
RETRIEVAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  recently-retrieved nil: NIL
  recently-retrieved t  : NIL

(GOAL RETRIEVAL)

>>> actr.buffer_status('goal','retrieval')
GOAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
RETRIEVAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
```

```
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  recently-retrieved nil: NIL
  recently-retrieved t  : NIL
['GOAL', 'RETRIEVAL']
```

ACT-R Software Interface

Before describing the specific functions that were used to have ACT-R call a function we had written for collecting a key press response, we will first describe the general mechanism which makes this possible at a fairly high level. The underlying details, which would be necessary if one wanted to extend things to create an interface to a different language, are well beyond the scope of the tutorial, but are available in a manual called remote in the docs directory of the distribution.

The key feature of the current ACT-R software which allows for the communication between ACT-R and arbitrary 'other' code is that it has been built around a central RPC (remote procedure call) system. The central RPC system (which we will refer to as the dispatcher) is responsible for accepting connections from clients (which include the core ACT-R code), maintaining the set of commands which those clients have made available, and coordinating the communication between a client that wants to execute a command and the client which has provided that command. The clients can be connected to the dispatcher directly through Lisp (as is the case for the core ACT-R code and any code loaded directly into the Lisp running ACT-R) or through a TCP/IP socket connection to the dispatcher (which is how the Python and ACT-R Environment connections are made), and the dispatcher allows for an unlimited number of clients to be connected at any time (theoretically at least since there are of course computational constraints). Any of the connected clients can add a command to the set available from the dispatcher, and that command can then be used by any of the connected clients with the dispatcher responsible for handling the communication between them.

In addition to supporting the communication between clients, the dispatcher also provides a 'monitoring' mechanism through which any command which has been added can get called automatically when another command is used. This monitoring mechanism allows a client to provide commands which other clients can then detect and respond to without that original client needing to know about every other client that wants to be notified. For example, this is why the output of the model trace is shown in both an interactive Python session and the ACT-R window – both of those clients are monitoring the commands responsible for printing the trace and then displaying the results.

Both the Lisp and Python interfaces to ACT-R provide the modeler with access to that central dispatcher. That allows the modeler to add new commands which can be called by code in ACT-R (which we will see in unit 5 when providing a function to create similarity values) and which can monitor other commands, as is done in this unit to

monitor the output-key command provided by the default keyboard device. Here we will describe the functions that provide access to the dispatcher.

To add a new command one uses the **add-act-r-command** or **add_command** function. It requires one parameter and has four optional parameters (only two of which will be described here). The first parameter must be a string which is the name of the new command to add. That name is case sensitive and it must not match the name of a command which already exists. The second parameter is optional, and specifies the local function which should be called when that command is evaluated (if no command is indicated then there is no activity associated with that command but it can still be called and monitored). The third parameter is also optional, but if given should be a string which provides some documentation about the command being added. If the command is added successfully then the function returns a true result (t or True) and if not, a warning is displayed and a null result (nil or False) is returned.

Once a command has been added it can be removed using the **remove-act-r-command** or **remove_command** function. That requires a single parameter which is the string that names a command. If it is successfully removed a true result is returned and if not a null result is returned.

To monitor a command the **monitor-act-r-command** or **monitor_command** function is used. It requires two parameters. The first parameter should be a string that names a command available from the dispatcher. That is the command which is being monitored. The second parameter should be a string which names another command available from the dispatcher. That is the command which is monitoring the other. The monitoring command will be called after every call to the monitored command and it will be passed the same parameters which the monitored command was given. If the monitoring is set up successfully then a true result will be returned and if not a null result will be returned.

To stop monitoring a command the **remove-act-r-command-monitor** or **remove_command_monitor** function is used. It requires two parameters which are the same as were specified when monitoring was initiated. The first is a string which names the command to monitor and the second is a string naming the command which is currently monitoring it but should stop monitoring now. If the monitoring is successfully removed then a true result is returned and if not a null result is returned.