Unit 2 Code Description

From this point on in the tutorial most of the models will have accompanying files that implement the experiment for the model to perform. The experiments are implemented in both Lisp and Python, and either version can be used with the model – the operation of the model does not depend on which language is used to create the experiment. The files with the code for the experiments contain comments describing how they work. In the code text for the units we will provide additional details about using ACT-R, the ACT-R commands used in creating the experiments, and information about how the models interact with the experiments.

More ACT-R Parameters

Before describing the experiments, we will first provide some information about the new parameters which were used in the models for this unit. All of the new parameters were used in the demo2 model and were set like this:

```
(sgp :seed (123456 0))
(sgp :v t :show-focus t :trace-detail high)
```

The first **sgp** command is used to set the :seed parameter. This parameter controls the starting point for the pseudo-random number generator used by ACT-R. Typically you do not need to use this parameter; however by setting it to a fixed value the model will always produce the same behavior (assuming that all the variation is attributable to randomness generated using the ACT-R mechanisms). That is why the randomly chosen letter for the demo2 task was always "V". If you remove this parameter setting from the model, save it, and then reload, you will see different letters chosen when the experiment is run. For the tutorial models, we will often set the :seed parameter in the demonstration model of a unit so that the model always produces exactly the same trace as presented in the unit text, but you should feel free to remove that to further investigate the models.

The second **sgp** call sets three parameters. The :v (verbose) parameter controls whether the trace of the model is output. If :v is **t** (which is the default value) then the trace is displayed and if :v is set to **nil** the trace is not printed. It is also possible to direct the trace to an external file instead of the having it printed in the interface, and you should consult the reference manual for information on how to do that. When the trace is not printed the model can run significantly faster, and that will be important in later units when we are running the models through the experiments multiple times to collect data. The :show-focus parameter controls whether or not the visual attention ring is displayed in the experiment window when the model is performing the task. It is a useful debugging tool, but for some displays you may not want it because it could obscure other things you want to see. If it is set to the value **t** then the red ring will be displayed. If it is set to **nil** then it will not be shown. It can also be set to the name of a color e.g. green, blue, yellow, etc. to change how it is displayed which can be helpful when there are multiple models simultaneously interacting with the same task. The :trace-detail

parameter, which was described in the unit 1 code description document, is set to control how much information is shown in the model's trace, and with a value of high all of the actions of the modules are shown.

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 custom visual feature information 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 that simulated clock can pause arbitrarily long before advancing to allow for instantaneous presentations in that simulated time space, and similarly, it has no latency on detecting the model's 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

Now that we have introduced the use of Python with ACT-R in the tutorial, we will show how some of the ACT-R commands which the unit 1 code document showed being used at the ACT-R (Lisp) prompt can also be used 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 an implementation of the remote interface described later in 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[1]. 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, reset, run for .1 seconds, and then reloaded.

```
>>> import actr
ACT-R connection has been started.
>>> 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 FIVE
     0.100   DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL FIVE
     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 ZERO
     0.200   DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL ZERO
     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 SIX
     0.300   DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL SIX
     0.300   PROCEDURAL           CONFLICT-RESOLUTION
     0.350   PROCEDURAL           PRODUCTION-FIRED INCREMENT-SUM
     0.350   PROCEDURAL           CLEAR-BUFFER RETRIEVAL
```

1 The Python module included with the tutorial is sufficient for running the tasks included with the tutorial. It does not contain functions for accessing all of the commands available through the ACT-R remote interface, and it has some assumptions about how it will be used based on the needs of the tutorial. For more complex tasks or where performance is of primary importance, one might be better served by creating a custom interface instead of using the one that is built for the tutorial.

```
     0.350   DECLARATIVE           start-retrieval
     0.350   PROCEDURAL            CONFLICT-RESOLUTION
     0.400   DECLARATIVE           RETRIEVED-CHUNK ONE
     0.400   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL ONE
     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 SEVEN
     0.500   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL SEVEN
     0.500   PROCEDURAL            CONFLICT-RESOLUTION
     0.550   PROCEDURAL            PRODUCTION-FIRED TERMINATE-ADDITION
SEVEN
     0.550   PROCEDURAL            CLEAR-BUFFER RETRIEVAL
     0.550   PROCEDURAL            CONFLICT-RESOLUTION
     0.550   ------                Stopped because no events left to process
[0.55, 77, None]
>>> actr.reset()
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 FIVE
     0.100   DECLARATIVE           SET-BUFFER-CHUNK RETRIEVAL FIVE
     0.100   PROCEDURAL            CONFLICT-RESOLUTION
     0.100   ------                Stopped because time limit reached
[0.1, 20, None]
>>> actr.reload()
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 chunks from the **goal** and **visual** buffers at the end of running the addition model from unit 1:

```
? (buffer-chunk goal visual)
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   ARG1  FIVE
   ARG2  TWO
   SUM  SEVEN

VISUAL: NIL
(GOAL-CHUNK0 NIL)
```

From Python however we must specify the arguments that are names (goal and visual in this case) as strings. Here is the corresponding use of actr.buffer_chunk:

```
>>> actr.buffer_chunk('goal','visual')
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   ARG1  FIVE
```

```
   ARG2  TWO
   SUM  SEVEN

VISUAL: NIL
['GOAL-CHUNK0', None]
```

The other things to notice are that the name returned in the list is also a string and that an empty buffer is reported as None in the list returned.

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). Here are examples of getting all of the chunks in DM and of only printing specific ones using the addition model from unit 1:

```
>>> actr.dm()
SECOND-GOAL
   ARG1  FIVE
   ARG2  TWO

TEN
   NUMBER  TEN

NINE
   NUMBER  NINE
   NEXT  TEN

EIGHT
   NUMBER  EIGHT
   NEXT  NINE

SEVEN
   NUMBER  SEVEN
   NEXT  EIGHT

SIX
   NUMBER  SIX
   NEXT  SEVEN

FIVE
   NUMBER  FIVE
   NEXT  SIX

FOUR
   NUMBER  FOUR
   NEXT  FIVE

THREE
   NUMBER  THREE
   NEXT  FOUR

TWO
   NUMBER  TWO
   NEXT  THREE

ONE
   NUMBER  ONE
   NEXT  TWO

ZERO
   NUMBER  ZERO
   NEXT  ONE
```

```
['SECOND-GOAL', 'TEN', 'NINE', 'EIGHT', 'SEVEN', 'SIX', 'FIVE', 'FOUR', 'THREE',
'TWO', 'ONE', 'ZERO']
>>> actr.dm('one','three')
ONE
   NUMBER  ONE
   NEXT  TWO

THREE
   NUMBER  THREE
   NEXT  FOUR

['ONE', 'THREE']
```

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 addend1 slot using the tutor-model-solution model from unit 1:

```
>>> actr.sdm('-','addend1','1')
GOAL
   ONE1  6
   TEN1  3
   ONE2  7
   TEN2  4

FACT34
   ADDEND1  3
   ADDEND2  4
   SUM  7

FACT67
   ADDEND1  6
   ADDEND2  7
   SUM  13

FACT103
   ADDEND1  10
   ADDEND2  3
   SUM  13

['GOAL', 'FACT34', 'FACT67', 'FACT103']
```

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). Here is an example after running the addition model from unit 1 for .3 seconds:

```
>>> 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 ZERO
   +RETRIEVAL>
```

```
       NUMBER =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>
       NUMBER =COUNT
       NEXT =NEWCOUNT
 ==>
   =GOAL>
       COUNT =NEWCOUNT
   +RETRIEVAL>
       NUMBER =SUM
)
It fails because:
The value in the NUMBER 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)
```

Another inspection command

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 the main unit 2 text). 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 visual)
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
VISUAL:
```

```
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  preparation free      : T
  preparation busy      : NIL
  processor free        : T
  processor busy        : NIL
  execution free        : T
  execution busy        : NIL
  last-command          : NONE
  scene-change          : NIL
  scene-change-value    : NIL
(GOAL VISUAL)

>>> actr.buffer_status('goal','visual')
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
VISUAL:
  buffer empty          : T
  buffer full           : NIL
  buffer failure        : NIL
  buffer requested      : NIL
  buffer unrequested    : NIL
  state free            : T
  state busy            : NIL
  state error           : NIL
  preparation free      : T
  preparation busy      : NIL
  processor free        : T
  processor busy        : NIL
  execution free        : T
  execution busy        : NIL
  last-command          : NONE
  scene-change          : NIL
  scene-change-value    : NIL
['GOAL', 'VISUAL']
```

ACT-R Software Interface

Before describing how the model interacts with the experiment, we will first describe at a very high level how the ACT-R software provides an interface for connecting to external systems like Python and the ACT-R Environment. [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 and there are simple examples for several languages in the examples/connections directory.] 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 also includes the ACT-R code in Lisp), 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 information 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 anything connected to the dispatcher and which can be used to monitor other commands (which will be done in this unit to detect when a key has been pressed).

Adding commands

To add a new command one uses the **add-act-r-command** or **add_command** function. It requires one parameter and has five 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.

Removing commands

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.

Monitoring commands

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 first one. The monitoring command will be called after every call to the monitored command. 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.

Experiment Code

When writing the experiments for the tutorial 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 provided 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 should be easier to see the similarities and differences between the human and model versions of the tasks.

Most of the experiments for the tutorial have a simple structure: some initialization is performed for the task (and possibly the model), a stimulus is presented, and then a response is recorded from the participant. That process may be repeated multiple time with the collected responses then being aggregated in some way to produce the data for the task. Given that basic task description, we will now describe the ACT-R commands used to implement those processes in the simple tasks for this unit.

Initial setup

Loading a model

The first thing that almost all of the tutorial experiment programs do is load the corresponding model for the task from the tutorial. That way one does not need to load two different files, and it makes sure that the appropriate model gets loaded. That

typically happens with a call like this using the **load-act-r-model/load_act_r_model** function (these are from the demo2 experiment):

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

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

Note that this does assume that the ACT-R tutorial files are located in their original directory with the ACT-R software. If you have moved the tutorial materials, or you want to load a different model file you can still do so using either that function at the prompt or by using the "Load ACT-R code" button in the Environment.

Creating the stimuli

Often one will want to randomize stimuli for a task in some way, and there are some ACT-R functions which can be used to help with that.

The **permute-list/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.

The **act-r-random/actr.random** function can be used to return a pseudo-random number[2]. It requires one parameter which must be a positive 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.

While similar functions are typically already available in most languages, using the randomizing functions provided by ACT-R when creating tasks allows one to have the model and task using the same generator for the "random" sequence of events which means that setting a single initial random seed value can be used to repeat the exact same sequence of events. That can can be very useful when trying to determine why a model is not working because the same situation can be run over again if one knows the starting seed, and it is also useful when creating demonstrations or examples (like in the tutorial) because it guarantees that a model will produce the same trace every time they are run in a task as long as the :seed parameter is set.

Stimuli Presentation

For presenting the tasks in the tutorial we will be using the AGI (described above). That will involve creating a window to display the task, and then displaying the stimuli in that window. In this unit that will only involve text items and these are the commands necessary.

creating a window

The **open-exp-window/open_exp_window** function is used to open an AGI 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,

2 The specific algorithm used for the ACT-R random numbers is currently the MT19937 generator.

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 (since multiple windows can be open at once) and it is also a valid device list that can be installed for the model to interact with (shown below). There are also several other parameters which may be provided when creating a window and those will be described in later units.

displaying text in a window

The **add-text-to-exp-window/add_text_to_exp_window** function will display text in an AGI 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/clear_exp_window** function is used to clear all items from an AGI 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.

checking for a visible window

As a safety check in the tutorial tasks, an additional AGI function is used to make sure that there is a visible window available for a person to perform the task. The **visible-virtual-available?/visible_virtual_available** function is used to test whether the AGI was able to open a visible window. If so, then the functions return a true value otherwise they return a false value. If the ACT-R Environment application is running then visible windows can be opened. There is another visible virtual window tool available which works in a browser that can be used (instead of or in addition to the Environment), and it is also possible to create your own window handler for the AGI, but those are not described in the tutorial.

Model interaction with tasks

devices

To have a model interact with a task it must be told which 'devices' to use by using the **install-device/install_device** function. That function requires one parameter which must

be a specification of a device for the model. A device is the term used for an interface that has been created for one of ACT-R's perceptual or motor modules to interact with (like a keyboard, mouse, microphone, etc). 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 automatically install virtual keyboard and mouse devices which the model can use 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.

key presses

When a key is pressed in an AGI 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 a key press the event is also shown in the trace as seen in the last line of this segment of the trace for performing the press-key action during the demo2 task:

```
0.485   PROCEDURAL              PRODUCTION-FIRED RESPOND
0.485   PROCEDURAL              MOD-BUFFER-CHUNK GOAL
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 0.485
0.735   PROCEDURAL              CONFLICT-RESOLUTION
0.785   MOTOR                   INITIATION-COMPLETE 0.485
0.785   PROCEDURAL              CONFLICT-RESOLUTION
0.885   KEYBOARD                output-key DEMO2 v
```

To detect and record key presses in an AGI window one will need to monitor the **output-key** command to be notified when the **output-key** action happens.

running a model

The **run** function was described in unit 1 as requiring one parameter which specifies a time limit for how long to run the model. It also takes an optional second parameter. The **run** function causes the simulated clock for the ACT-R system to advance which allows 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. When the model is interacting with a real window it should always be run in step with real time to ensure that the display is updated appropriately as the model performs the task, but if it is using a virtual window it does not need to run in real time.

running a person

When a waiting loop is needed to collect a real response from a person in an AGI window the **process-events/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 ACT-R provides an opportunity for other threads/processes to run which may be necessary for the system to be able to handle the interaction and it also allows ACT-R the opportunity to call any monitoring function(s) in response to a user's input.

A while Lisp macro provided with ACT-R

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.

Final technical programming safety note

One important thing to note about the code for these experiments is that the functions that are monitoring key presses 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. Since both threads are accessing the same global variable(s) in these experiment implementations 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 typically looking for in the tutorial tasks 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 these tasks, and that protection has been ignored for the purpose of keeping the tutorial tasks easier to read. If you are creating more complicated tasks which are using monitors for things like key presses and mouse clicks, or any 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 (which is a programming issue well beyond the scope of this tutorial).