Unit 4 Code Description

There are only a couple of new commands used in the models for this unit and one of them, set-all-base-levels, was discussed in the unit text. So there is not really much of anything new to discuss about the functions used to run the models, but there is a different approach to running the models used for the zbrodoff experiment compared to the paired associate task and all of the previous units' tasks. We will start by describing the differences between those approaches, and then look at the code which implements them with the focus more on the overall structure of the experiment implementations and model interaction than the details of every function call.

So far you have seen what can be described as an iterative or "trial at a time" approach to writing the experiments for models. The experiments run by executing some setup code for a trial, running the model to completion on that trial, recording a result and then repeating that process for the next trial until all the trials are done. That style is a commonly used approach, but it has some drawbacks which you may have encountered. For instance, when running an experiment it is not possible to terminate the experiment using either the Stop button on the ACT-R Control Panel or the one in the Stepper tool. Instead, you have to break out of the execution of the experiment function from the system that is running it (often Control-C will work to interrupt things, but that can vary for different interfaces). That is because the Stop buttons only stop the current "run" of ACT-R, but they cannot affect the code which is making those calls to run things. For models with few trials or that get reset on each trial that may not be too big of a problem, but for large experiments or models that need to learn from trial to trial that can make things difficult to work with, particularly if there is a problem with the model on a later trial that forces one to abandon a very long run by terminating the experiment code which likely doesn't have any way to continue where the model left off.

An alternative way to write the experiments is with a more "event-driven" approach. The system which runs the ACT-R models is a general discrete-event simulation system which can be used to run other things as well, like an experiment for a model (or a person if precision timing information isn't required). Calling one of the ACT-R running commands causes all of the events which have been created to be executed in either a simulated time or real time sequence. Up to now those events have been mostly generated by the model, e.g. production firing, memory retrieval, and key presses, or by ACT-R commands like goal-focus. However, as was seen in the sperling task, arbitrary events can also be scheduled to execute at particular simulated times. One can also use the interface events generated by the model (like output-key) to do things other than just record the response. By scheduling events to occur at appropriate times and putting some of the control into the functions that handle the model's actions the experiment can run "with" the model instead of "around" it. Such an experiment only needs to call the run function one time to complete the whole experiment instead of once (or more) per trial.

Having the model running in an event-driven experiment with a single call to run typically allows for more interactive control of the task as a whole. The Stepper tool in the Environment will also pause on user scheduled events and the Environment's Stop buttons will stop the whole experiment if it is being driven by the events that are run. That allows

one to see exactly what is happening at specific points in the experiment without having to abort the experiment function. Additionally, to continue after stopping all one needs to do is then call run again to have the model and the experiment continue from where they left off since the events are still scheduled to occur at the appropriate times. It can also make writing the model itself easier because one doesn't have to make sure that the model "knows" when to stop for the task code to update and can just focus on having it respond to the events that occur as they occur instead of as a sequence of separate interactions.

The two models for this unit use those two different approaches. The paired associate task is written using the iterative approach, and the zbrodoff task is written as an event-driven experiment.

Paired associate task

Lisp

We start by loading the corresponding model for the task.

```
(load-act-r-model "ACT-R:tutorial;unit4;paired-model.lisp")
```

Then we define some global variables to hold the key pressed, the time of the response, the possible stimuli to use for the experiment, and the data from the original experiment for comparison.

```
(defvar *response* nil)
(defvar *response-time* nil)

(defvar *pairs* '(("bank" "0") ("card" "1") ("dart" "2") ("face" "3") ("game" "4")
                  ("hand" "5") ("jack" "6") ("king" "7") ("lamb" "8") ("mask" "9")
                  ("neck" "0") ("pipe" "1") ("quip" "2") ("rope" "3") ("sock" "4")
                  ("tent" "5") ("vent" "6") ("wall" "7") ("xray" "8") ("zinc" "9")))

(defvar *paired-latencies* '(0.0 2.158 1.967 1.762 1.680 1.552 1.467 1.402))
(defvar *paired-probability* '(0.000 .526 .667 .798 .887 .924 .958 .954))
```

The paired-task function takes two required parameters which are the number of pairs to present in a trial and the number of trials to run. It also takes an optional parameter which can be specified as true to run a person through the task instead of the model.

```
(defun paired-task (size trials &optional human)
```

It monitors output-key with the respond-to-key-press function to record the responses.

```
  (add-act-r-command "paired-response" 'respond-to-key-press
                     "Paired associate task key press response monitor")
  (monitor-act-r-command "output-key" "paired-response")
```

It calls do-experiment to run the actual experiment, removes the monitoring functions when it's done, and returns the result of the do-experiment call.

```
  (prog1
```

```
    (do-experiment size trials human)

  (remove-act-r-command-monitor "output-key" "paired-response")
  (remove-act-r-command "paired-response")))
```

The respond-to-key-press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press (or nil if the key was pressed by a person interacting with the window) and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get-time function, and we can pass the model parameter to get-time to get the appropriate time since a model name is a true value which results in the ACT-R time being returned whereas if it is a person making the response the model value will be nil which means get-time will use the real clock instead of the ACT-R clock.

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

  (setf *response-time* (get-time model))
  (setf *response* key))
```

The do-experiment function takes three parameters which are the number of pairs to present in a trial, the number of trials to run, and whether it is a person or model doing the task. It runs the experiment for the size and number of trials requested and returns a list of lists. There is one sublist for each of the trials and they are in the order of presentation. Each of the sublists contains the percentage of answers correct and the average response time for the correct answers in the corresponding trial.

```
(defun do-experiment (size trials human)
```

First reset the ACT-R system and model to initial state.

```
  (reset)
```

Create variables to hold the list of results, an indication of whether the model is doing the task, and the current task window which is opened immediately with a visible window for a person and virtual for the model.

```
  (let* ((result nil)
         (model (not human))
         (window (open-exp-window "Paired-Associate Experiment" :visible human)))
```

If the model is doing the task tell it to interact with that window.

```
    (when model
      (install-device window))
```

Loop over the number of trials indicated.

```
(dotimes (i trials)
```

Create variables to hold the count and times for correct responses during this trial.

```
  (let ((score 0.0)
        (time 0.0))
```

Loop over the required number of pairs chosen randomly from *pairs*.

```
    (dolist (x (permute-list (subseq *pairs* (- 20 size))))
```

Clear the window and display the prompt.

```
      (clear-exp-window window)
      (add-text-to-exp-window window (first x) :x 150 :y 150)
```

Clear the *response* variable and record the current time.

```
      (setf *response* nil)
      (let ((start (get-time model)))
```

If it's the model run it for exactly 5 seconds, or if it's a person loop until 5000 milliseconds have passed making sure to call process-events in the loop.

```
        (if model
            (run-full-time 5)
          (while (< (- (get-time nil) start) 5000)
            (process-events)))
```

If the response is correct update the score and time values.

```
        (when (equal *response* (second x))
          (incf score 1.0)
          (incf time (- *response-time* start)))
```

Clear the window, display the associated number, and record the current time.

```
        (clear-exp-window window)
        (add-text-to-exp-window window (second x) :x 150 :y 150)
        (setf start (get-time model))
```

If it's the model run it for exactly 5 seconds, or if it's a person loop until 5000 milliseconds have passed making sure to call process-events in the loop.

```
        (if model
            (run-full-time 5)
          (while (< (- (get-time nil) start) 5000)
```

```
            (process-events)))))
```

Save the proportion correct and the average time for a correct response.

```
        (push (list (/ score size) (if (> score 0) (/ time score 1000.0) 0)) result)))
```

Return the list of trial results in order (reversed since the values were pushed onto the front as the trials progressed).

```
    (reverse result)))
```

The paired-experiment function takes one parameter which is the number of times to repeat the full experiment (20 pairs and 8 trials). The experiment is run that many times and the results are averaged and compared to the experimental results.

```
(defun paired-experiment (n)
  (let ((data nil))
    (dotimes (i n)
      (if (null data)
          (setf data (paired-task 20 8))
        (setf data (mapcar (lambda (x y)
                             (list (+ (first x) (first y))
                                   (+ (second x) (second y))))
                           data (paired-task 20 8)))))
    (output-data data n)))
```

The output-data function takes two parameters. The first is a list of cumulative data from running multiple iterations of the experiment and the second parameter indicates how many repetitions were added into that cumulative data. It averages that data and then calls print-results to display the comparison and table for both the latency and accuracy data.

```
(defun output-data (data n)
  (print-results (mapcar (lambda (x) (/ (second x) n)) data) *paired-latencies* "Latency")
    (print-results (mapcar (lambda (x) (/ (first x) n)) data) *paired-probability*
"Accuracy"))
```

The print-results function takes three parameters. The first is the list of data from running the experiment. The second is the experimental results for that data, and the third is the label to print when displaying that data. The new data is compared to the experimental results and printed along with a table of the new data.

```
(defun print-results (predicted data label)
 (format t "~%~%~A:~%" label)
  (correlation predicted data)
  (mean-deviation predicted data)
  (format t "Trial   1      2      3      4      5      6      7      8~%")
  (format t "     ~{~8,3f~}~%" predicted))
```

Python

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

```
import actr
```

Then it loads the corresponding model for the task.

```
actr.load_act_r_model("ACT-R:tutorial;unit4;paired-model.lisp")
```

Define some global variables to hold the key pressed, the time of the response, the possible stimuli to use for the experiment, and the data from the original experiment for comparison.

```
response = False
response_time = False

pairs = list(zip(['bank','card','dart','face','game','hand','jack','king','lamb','mask',
                  'neck','pipe','quip','rope','sock','tent','vent','wall','xray','zinc'],
                 ['0','1','2','3','4','5','6','7','8','9',
                  '0','1','2','3','4','5','6','7','8','9']))

latencies = [0.0, 2.158, 1.967, 1.762, 1.680, 1.552, 1.467, 1.402]
probabilities = [0.0, .526, .667, .798, .887, .924, .958, .954]
```

The task function takes two required parameters which are the number of pairs to present in a trial and the number of trials to run. It also takes an optional parameter which can be specified as true to run a person through the task instead of the model.

```
def task (size,trials,human=False):
```

It monitors output-key with the respond_to_key_press function to record the responses.

```
    actr.add_command("paired-response",respond_to_key_press,
                     "Paired associate task key press response monitor")
    actr.monitor_command("output-key","paired-response")
```

It calls do_experiment to run the actual experiment, removes the monitoring functions when it's done, and returns the result of the do_experiment call.

```
    result = do_experiment(size,trials,human)

    actr.remove_command_monitor("output-key","paired-response")
    actr.remove_command("paired-response")

    return result
```

The respond_to_key_press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press (or None if the key was pressed by a person interacting with the window) and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get_time function, and we can pass the model parameter to get_time to get the appropriate time since a model name is a true value which results in the ACT-R time being returned whereas if it is a person making the response the model value will be None which means get_time will use the real clock instead of the ACT-R clock.

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

    response_time = actr.get_time(model)

    response = key
```

The do_experiment function takes three parameters which are the number of pairs to present in a trial, the number of trials to run, and whether it is a person or model doing the task. It runs the experiment for the size and number of trials requested and returns a list of tuples. There is one tuple for each of the trials and they are in the order of presentation. Each of the tuples contains the percentage of answers correct and the average response time for the correct answers in the corresponding trial.

```
def do_experiment(size, trials, human):
```

First reset the ACT-R system and model to initial state.

```
    actr.reset()
```

Create variables to hold the list of results, an indication of whether the model is doing the task, and the current task window which is opened immediately with a visible window for a person and virtual for the model.

```
    result = []
    model = not(human)
    window = actr.open_exp_window("Paired-Associate Experiment", visible=human)
```

If the model is doing the task tell it to interact with that window.

```
    if model:
        actr.install_device(window)
```

Loop over the number of trials indicated.

```
    for i in range(trials):
```

Create variables to hold the count and times for correct responses.

```
        score = 0
        time = 0
```

Loop over the required number of randomly chosen pairs.

```
        for prompt,associate in actr.permute_list(pairs[20 - size:]):
```

Clear the window and display the prompt.

```
    actr.clear_exp_window(window)
    actr.add_text_to_exp_window (window, prompt, x=150 , y=150)
```

Clear the response variable and record the current time.

```
    global response
    response = ''
    start = actr.get_time(model)
```

If it’s the model run it for exactly 5 seconds, or if it’s a person loop until 5000 milliseconds have passed making sure to call process_events.

```
    if model:
        actr.run_full_time(5)
    else:
        while (actr.get_time(False) - start) < 5000:
            actr.process_events()
```

If the response is correct update the score and time values.

```
    if response == associate:
        score += 1
        time += response_time - start
```

Clear the window, display the associated number, and record the current time.

```
    actr.clear_exp_window(window)
    actr.add_text_to_exp_window (window, associate, x=150 , y=150)
    start = actr.get_time(model)
```

If it’s the model run it for exactly 5 seconds, or if it’s a person loop until 5000 milliseconds have passed making sure to call process_events.

```
    if model:
        actr.run_full_time(5)
    else:
        while (actr.get_time(False) - start) < 5000:
            actr.process_events()
```

Save the proportion correct and the average time for a correct response.

```
if score > 0:
    average_time = time / score / 1000.0
else:
    average_time = 0

result.append((score/size,average_time))
```

Return the list of trial results.

```
    return result
```

The experiment function takes one parameter which is the number of times to repeat the full experiment (20 pairs and 8 trials). The experiment is run that many times and the results are collected and passed to output_data for comparison to the experimental results.

```
def experiment(n):

    for i in range(n):

        if i == 0:
            data = task(20,8)
        else:
            data = list(map(lambda x,y: (x[0] + y[0],x[1] + y[1]),data,task(20,8)))

    output_data(data,n)
```

The output_data function takes two parameters. The first is a list of cumulative data from running multiple iterations of the experiment and the second parameter indicates how many repetitions were added into that cumulative data. It averages that data and then calls print_results to display the comparison and table for both the latency and accuracy data.

```
def output_data(data,n):

    print_results(list(map(lambda x: x[1]/n,data)),latencies,"Latency")
    print_results(list(map(lambda x: x[0]/n,data)),probabilities,"Accuracy")
```

The print_results function takes three parameters. The first is the list of data from running the experiment. The second is the experimental results for that data, and the third is the label to print when displaying that data. The new data is compared to the experimental results and printed along with a table of the new data.

```
def print_results(predicted,data,label):

    print()
    print(label)
    actr.correlation(predicted,data)
    actr.mean_deviation(predicted,data)
    print("Trial    1       2       3       4       5       6       7       8")
    print("     ",end='')
    for i in predicted:
        print('%8.3f' % i,end='')
    print()
```

New Commands

The only new command used is run-full-time/run_full_time.

Run-full-time/run_full_time – this function takes one required parameter which is the

time to run a model in seconds and an optional parameter to indicate whether to run the model in step with real time. The model will run until the requested amount of time passes whether or not there is something for the model to do i.e. it guarantees that the model will be advanced by the requested amount of time. If the optional parameter is provided and is a true value, then the model is advanced in step with real time instead of being allowed to run as fast as possible in its own simulated time, and if a number is provided instead of True then that sets the scale to use for advancing model time relative to real time.

Zbrodoff task

The **zbrodoff** task is written using the event-driven approach and to run the model through the task the code will only need to call run once regardless of how many trials are to be run. This task also differs from all of the previous ones because the data to be collected has to be organized by block and the addend condition. There are many ways one could approach that, and for this task we are going to keep the collection simple and just record all of the trial data as it happens and then process that collection of data at the end to sort it into the appropriate cases. Alternatively, we could record it into an array or other data structure as it happens, but this choice was made to keep the components that are relevant to the event-driven approach easier to follow since that is the important aspect of this task for modeling purposes.

Lisp

It starts by loading the initial model for the task which counts through the alphabet to solve all of the problems.

```
(load-act-r-model "ACT-R:tutorial;unit4;zbrodoff-model.lisp")
```

Then it defines some global variables to hold the trials to be presented, the results that have been collected, and the data from the original experiment.

```
(defvar *trials*)
(defvar *results*)

(defvar *zbrodoff-control-data* '(1.84 2.46 2.82 1.21 1.45 1.42 1.14 1.21 1.17))
```

Because the functions to run this task use an optional parameter to indicate whether or not to show the task window, to keep things easier to use, this experiment uses a global variable to indicate whether it is a person or model doing the task. If it is set to **nil** then it runs a person and if it is set to **t** it runs the model.

```
(defparameter *run-model* t)
```

Instead of using lists to represent the information in a trial and to describe a response we create a custom structure to hold that data in a more descriptive format. A trial consists of a block number, the number addend which indicates the condition, the text to display on the screen, the correct answer, whether the window should be visible, whether the answer was correct, the time the trial started, and the response time for the trial.

```
(defstruct trial block addend text answer visible correct start time)
```

The construct-trial function takes a block number, a list of the items which describe the problem to present, and an optional parameter to indicate whether or not the window should be visible. It creates and returns a trial structure containing the appropriate information.

```
(defun construct-trial (block problem &optional visible)
  (destructuring-bind (addend1 addend2 sum answer) problem
    (make-trial :block block
                :addend addend2
                :text (format nil "~a + ~a = ~a" addend1 addend2 sum)
                :answer answer
                :visible visible)))
```

The present-trial function takes one parameter which is a trial structure and an optional parameter which indicates whether or not to open a new window for this trial (since this task is running continuously it will run faster if it uses the same window repeatedly, but because the same code is used to run it for a variety of different situations it needs to know when to start over with a new display).

```
(defun present-trial (trial &optional (new-window t))
```

If a new window is indicated then it opens one setting the visible status of the window based on the setting in the trial structure provided, and if it is running the model it installs that window device. If a new window is not indicated then it clears the current experiment window.

```
  (if new-window
          (let ((w (open-exp-window "Alpha-arithmetic Experiment" :visible (trial-visible
trial))))
        (when *run-model*
          (install-device w)))
    (clear-exp-window))
```

It then adds the text for this trial to the window and records the current time in the trial structure.

```
  (add-text-to-exp-window nil (trial-text trial) :x 100 :y 150)

  (setf (trial-start trial) (get-time *run-model*)))
```

The respond-to-key-press function will be set up to monitor the output-key actions, and thus will be called with two parameters when a key is pressed: the name of the model who pressed the key (or nil if it is a person) and the string naming the key that was pressed. Unlike the previous tasks, since this one is event-driven we will do more than just record the key and time in this function.

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

Remove the current trial from those being presented.

```
(let ((trial (pop *trials*)))
```

Set the response time and correctness of the response in that trial structure.

```
  (setf (trial-time trial) (/ (- (get-time *run-model*) (trial-start trial)) 1000.0))
  (setf (trial-correct trial) (string-equal (trial-answer trial) key))
```

Store that trial on the list of results.

```
  (push trial *results*))
```

If there are anymore trials to present then present the first one on the list and indicate that a new window is not needed. This is what makes this code event-driven – the event of pressing a key directly causes the presentation of the next trial.

```
(when *trials*
  (present-trial (first *trials*) nil)))
```

The collect-responses function takes one parameter which is the number of trials that are to be run.

```
(defun collect-responses (count)
```

The global list of results is cleared.

```
(setf *results* nil)
```

The respond-to-key-press function is setup to monitor the output-key command.

```
(add-act-r-command "zbrodoff-response" 'respond-to-key-press
                   "Zbrodoff task key press response monitor")
(monitor-act-r-command "output-key" "zbrodoff-response")
```

The first trial is presented in a new window.

```
(present-trial (first *trials*))
```

If the model is performing the task then it is run for up to 10 seconds times the number of trials that need to be collected (it is assumed that the model will respond in 10 seconds or less per trial on average).

```
(if *run-model*
    (run (* 10 count))
```

If a person is performing the task then wait for the appropriate number of responses to be recorded calling process-events in the loop.

```
  (while (< (length *results*) count)
    (process-events)))
```

Remove the output-key monitoring.

```
(remove-act-r-command-monitor "output-key" "zbrodoff-response")
(remove-act-r-command "zbrodoff-response"))
```

The zbrodoff-problem function takes four required parameters and one optional parameter. The four required parameters are the strings that represent the equation to present, for example "A" "2" and "C", and the string indicating the correct response – either "k" for correct or "d" for incorrect. The optional parameter controls whether the trial is shown in a visible window or not and defaults to the negation of whether or not the model is doing the task i.e. by default if the model is doing the task visible is **nil** in which case the window will be virtual and if a person is doing the task the window will be shown. Providing a value of **t** for the (optional) fifth parameter will cause the window to be displayed while the model is doing the task.

```
(defun zbrodoff-problem (addend1 addend2 sum answer &optional (visible (not *run-model*)))
```

This is only one trial, so set the list of trials to a list of only that one trial.

```
(setf *trials* (list (construct-trial 1 (list addend1 addend2 sum answer) visible)))
```

Call collect-responses to perform the task and then analyze the results.

```
(collect-responses 1)
(analyze-results))
```

The zbrodoff-set function takes one optional parameter which controls whether the window is shown or not, the same as the zbrodoff-problem function. It runs once through a random permutation of the set of equations where a set is two instances of each of the equations with the addends 2, 3, and 4 in each of the true and false conditions, which is a total of 24 problems.

```
(defun zbrodoff-set (&optional (visible (not *run-model*)))
  (setf *trials* (create-set 1 visible))
  (collect-responses 24)
  (analyze-results))
```

The zbrodoff-block function takes one optional parameter like the problem and set functions. It runs one block of the experiment, which is eight repetitions of the set of equations, or a total of 192 problems.

```
(defun zbrodoff-block (&optional (visible (not *run-model*)))
  (setf *trials* nil)
  (dotimes (i 8)
    (setf *trials* (append *trials* (create-set 1 visible))))
  (collect-responses 192)
  (analyze-results))
```

The zbrodoff-experiment function runs the whole experiment once, which is three full blocks, or a total of 576 trials. It takes two optional parameters. The first is to control

whether the window is shown or not as with the previous functions. The other controls whether the analysis is printed. The default is to have the analysis printed. Note that this function also calls reset to return the model to its initial condition. It is the only function in the experiment to do so. The other functions allow the model to maintain the information it has gained (which is the new chunks and the history of their use).

```
(defun zbrodoff-experiment (&optional (visible (not *run-model*)) (show t))
  (reset)
  (setf *trials* nil)
  (dotimes (j 3)
    (dotimes (i 8)
      (setf *trials* (append *trials* (create-set (+ j 1) visible)))))
  (collect-responses 576)
  (analyze-results show))
```

The zbrodoff-compare function takes one parameter, which is the number of times to run the whole experiment. It runs that many times through the experiment collecting the data which is then averaged and compared to the original experiment's results.

```
(defun zbrodoff-compare (n)
  (let ((results nil))
```

Run the experiment n times with a virtual window and without displaying the individual analysis of each run, and collect the results in a list.

```
    (dotimes (i n)
      (push (zbrodoff-experiment nil nil) results))
```

Compute the averages of the response times and the number of correct answers.

```
    (let ((rts (mapcar (lambda (x) (/ x (length results)))
                       (apply 'mapcar '+ (mapcar 'first results))))
          (counts (mapcar (lambda (x) (truncate x (length results)))
                          (apply 'mapcar '+ (mapcar 'second results)))))
```

Display the data fit between the current run and the experimental data and print the table of the results.

```
      (correlation rts *zbrodoff-control-data*)
      (mean-deviation rts *zbrodoff-control-data*)

      (print-analysis rts counts '(1 2 3) '("2" "3" "4") '(64 64 64)))))
```

The analyze-results function takes one optional parameter which controls whether or not the print the table of results. It computes the average response time and number of correct responses in the *results* global variable as a function of the number of blocks presented and the numerical addend and returns a list of those two results.

```
(defun analyze-results (&optional (show t))
  (let* ((blocks (sort (remove-duplicates (mapcar 'trial-block *results*)) '<))
         (addends (sort (remove-duplicates (mapcar 'trial-addend *results*)
                                           :test 'string-equal)
                        'string<))
```

```
        (counts nil)
        (rts nil)
        (total-counts (mapcar (lambda (x)
                                (/ (count x *results*
                                          :key 'trial-addend
                                          :test 'string=)
                                   (length blocks)))
                              addends)))

    (dolist (x blocks)
      (dolist (y addends)
        (let ((data (mapcar 'trial-time
                      (remove-if-not (lambda (z)
                                       (and (response-correct z)
                                            (string= y (response-addend z))
                                            (= x (response-block z))))
                                     *results*))))
          (push (length data) counts)
          (push (/ (apply '+ data) (max 1 (length data))) rts))))


    (when show
      (print-analysis (reverse rts) (reverse counts) blocks addends total-counts))

    (list (reverse rts) (reverse counts))))
```

The print-analysis function takes five parameters that describe a set of data for the task. The data is a list of response times and a list of corresponding correct answers. Then there are two lists that indicate the blocks and addend conditions represented in the data and finally a list of the total number of correct trials in each addend condition. Those data are then displayed in a table.

```
(defun print-analysis (rts counts blocks addends totals)
  (format t "~%        ")
  (dotimes (addend (length addends))
    (format t " ~6@a (~2d)" (nth addend addends) (nth addend totals)))
  (dotimes (block (length blocks))
    (format t "~%Block ~2d" (nth block blocks))
    (dotimes (addend (length addends))
      (format t " ~6,3f (~2d)" (nth (+ addend (* block (length addends))) rts)
        (nth (+ addend (* block (length addends))) counts))))
  (terpri))
```

The *data-set* variable is created to hold lists that represent all the problems to present in one set of the task.

```
(defvar *data-set* '(("a" "2" "c" "k")("d" "2" "f" "k")
                     ("b" "3" "e" "k")("e" "3" "h" "k")
                     ("c" "4" "g" "k")("f" "4" "j" "k")
                     ("a" "2" "d" "d")("d" "2" "g" "d")
                     ("b" "3" "f" "d")("e" "3" "i" "d")
                     ("c" "4" "h" "d")("f" "4" "k" "d")
                     ("a" "2" "c" "k")("d" "2" "f" "k")
                     ("b" "3" "e" "k")("e" "3" "h" "k")
                     ("c" "4" "g" "k")("f" "4" "j" "k")
                     ("a" "2" "d" "d")("d" "2" "g" "d")
                     ("b" "3" "f" "d")("e" "3" "i" "d")
                     ("c" "4" "h" "d")("f" "4" "k" "d")))
```

The create-set function takes two parameters. The first indicates a number to specify which block of the experiment is being performed and the second indicates whether or not the trials should be shown in a visible window. It returns a randomly ordered list of 24 trial structures that make up one set of the control condition of the experiment.

```
(defun create-set (block visible)
  (mapcar (lambda (x)
            (construct-trial block x visible))
    (permute-list *data-set*)))
```

Python

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

```
import actr
```

It loads the initial model for the task which counts through the alphabet to solve all of the problems.

```
actr.load_act_r_model("ACT-R:tutorial;unit4;zbrodoff-model.lisp")
```

Then it defines some global variables to hold the trials to be presented, the results that have been collected, and the data from the original experiment.

```
trials = []
results = []

control_data = [1.84, 2.46, 2.82, 1.21, 1.45, 1.42, 1.14, 1.21, 1.17]
```

Because the functions to run this task use an optional parameter to indicate whether or not to show the task window, to keep things easier to use, this experiment uses a global variable to indicate whether it is a person or model doing the task. If it is set to **False** then it runs a person and if it is set to **True** it runs the model.

```
run_model = True
```

Instead of using lists to represent the information in a trial and to describe a response we create a custom class to hold that data. A trial consists of a block number, the number addend which indicates the condition, the text to display on the screen, the correct answer, whether the window should be visible, whether the answer was correct, the time the trial started, and the response time for the trial.

```
class trial():
    def __init__(self,block,addend1,addend2,sum,answer,visible=None):
        self.block = block
        self.addend2 = addend2
        self.text = addend1 + " + " + addend2 + " = " + sum
        self.answer = answer.lower()
        if visible == None:
            self.visible = not(run_model)
        else:
            self.visible = visible
        self.correct = False
```

The present_trial function takes one parameter which is a trial object and an optional parameter which indicates whether or not to open a new window for this trial (since this task is running continuously it will run faster if it uses the same window repeatedly, but because the same code is used to run it for a variety of different situations it needs to know when to start over with a new display).

```
def present_trial(trial, new_window = True):
```

If a new window is indicated then it opens one setting the visible status of the window based on the setting in the trial structure provided, and if it is running the model it installs that window device. If a new window is not indicated then it clears the current experiment window.

```
    if new_window:
        w = actr.open_exp_window("Alpha-arithmetic Experiment", visible=trial.visible)
        if run_model:
          actr.install_device(w)
    else:
        actr.clear_exp_window()
```

It then adds the text for this trial to the window and records the current time in the trial structure.

```
    actr.add_text_to_exp_window(None,trial.text, x=100, y=150)

    trial.start = actr.get_time(run_model)
```

The respond_to_key_press function will be set up to monitor the output-key actions, and thus will be called with two parameters when a key is pressed: the name of the model who pressed the key (or None if it is a person) and the string naming the key that was pressed. Unlike the previous tasks, since this one is event-driven we will do more than just record the key and time in this function.

```
def respond_to_key_press (model,key):
    global trials,results
```

Set the response time and correctness of the response in the first trial.

```
    trials[0].time = (actr.get_time(run_model) - trials[0].start) / 1000.0

    if key.lower() == trials[0].answer :
        trials[0].correct = True
```

Store that trial on the list of results.

```
    results.append(trials[0])
```

Remove the current trial from those being presented.

```
    trials = trials[1:]
```

If there are anymore trials to present then present the first one on the list and indicate that a new window is not needed. This is what makes this code event-driven – the event of pressing a key directly causes the presentation of the next trial.

```
if len(trials) > 0 :
    present_trial(trials[0],False)
```

The collect_responses function takes one parameter which is the number of trials that are to be run.

```
def collect_responses(count):
```

The global list of results is cleared.

```
global results

results = []
```

The respond_to_key_press function is setup to monitor the output-key command.

```
actr.add_command("zbrodoff-response", respond_to_key_press,
                 "Zbrodoff task key press response monitor")
actr.monitor_command("output-key","zbrodoff-response")
```

The first trial is presented in a new window.

```
present_trial(trials[0])
```

If the model is performing the task then it is run for up to 10 seconds times the number of trials that need to be collected (it is assumed that the model will respond in 10 seconds or less per trial on average).

```
if run_model :
    actr.run(10 * count)
```

If a person is performing the task then wait for the appropriate number of responses to be recorded calling process_events in the loop.

```
else:
    while len(results) < count:
        actr.process_events()
```

Remove the output-key monitoring.

```
actr.remove_command_monitor("output-key","zbrodoff-response")
actr.remove_command("zbrodoff-response")
```

The problem function takes four required parameters and one optional parameter. The four required parameters are the strings that represent the equation to present, for example ‘A’ ‘2’ and ‘C’, and the string indicating the correct response – either ‘k’ for correct or ‘d’ for incorrect. The optional parameter controls whether the trial is shown in a visible

window or not and with the default value of **None** if the model is doing the task the window will be virtual and if a person is doing the task the window will be shown. Providing a value of **True** for the (optional) fifth parameter will cause the window to be displayed while the model is doing the task.

```
def problem(addend1,addend2,sum,answer,visible=None):
```

This is only one trial, so set the list of trials to a list of only that one trial.

```
    global trials
    trials = [trial(1,addend1,addend2,sum,answer,visible)]
```

Call collect_responses to perform the task and then analyze the results.

```
    collect_responses(1)
    return analyze_results()
```

The set function takes one optional parameter which controls whether the window is shown or not, the same as the problem function. It runs once through a random permutation of the set of equations where a set is two instances of each of the equations with the addends 2, 3, and 4 in each of the true and false conditions, which is a total of 24 problems.

```
def set(visible=None):

    global trials
    trials = create_set(1,visible)

    collect_responses(24)
    return analyze_results()
```

The block function takes one optional parameter like the problem and set functions. It runs one block of the experiment, which is eight repetitions of the set of equations, or a total of 192 problems.

```
def block(visible=None):
    global trials
    trials = []

    for i in range(8):
        trials = trials + create_set(1,visible)

    collect_responses(192)
    return analyze_results()
```

The experiment function runs the whole experiment once, which is three full blocks, or a total of 576 trials. It takes two optional parameters. The first is to control whether the window is shown or not as with the previous functions. The other controls whether the analysis is printed. The default is to have the analysis printed. Note that this function also calls reset to return the model to its initial condition. It is the only function in the experiment to do so. The other functions allow the model to maintain the information it has gained (which is the new chunks and the history of their use).

```
def experiment(visible=None,show=True):

    actr.reset()

    global trials
    trials = []

    for j in range(3):
        for i in range(8):
            trials = trials + create_set(j+1,visible)

    collect_responses(576)
    return analyze_results(show)
```

The compare function takes one parameter, which is the number of times to run the whole experiment. It runs that many times through the experiment collecting the data which is averaged and compared to the original experiment's results.

```
def compare(n):
```

Initialize some lists to hold the average data.

```
    rts = [0,0,0,0,0,0,0,0,0]
    counts = [0,0,0,0,0,0,0,0,0]
```

Run the experiment n times with a virtual window and without displaying the individual analysis of each run, and collect the results in the data lists.

```
    for i in range(n):
        r,c = experiment(False,False)
        rts = list(map(lambda x,y: x + y,rts,r))
        counts = list(map(lambda x,y: x + y,counts,c))
```

Compute the averages of the response times and the counts.

```
    rts = list(map(lambda x: x/n,rts))
    counts = list(map(lambda x: x/n,counts))
```

Display the data fit between the current run and the experimental data and print the table of the results.

```
    actr.correlation(rts,control_data)
    actr.mean_deviation(rts,control_data)

    print_analysis(rts,counts,[1,2,3],['2','3','4'], [192,192,192])
```

The analyze_results function takes one optional parameter which controls whether or not to print the table of results. It computes the average response time and number of correct responses in the results global variable as a function of the number of blocks presented and the numerical addend and returns a list of those two results.

```
def analyze_results(show=True):

    blocks = []
```

```
    addends = []
    data = dict()
    totals = dict()

    for i in results:
        if i.addend2 in totals:
            totals[i.addend2] += 1
        else:
            totals[i.addend2] = 1

        if i.correct:
            if (i.block,i.addend2) in data:
                data[(i.block,i.addend2)].append(i.time)
            else:
                data[(i.block,i.addend2)]=[i.time]
                if i.block not in blocks:
                    blocks.append(i.block)
                if i.addend2 not in addends:
                    addends.append(i.addend2)


    blocks.sort()
    addends.sort()

    rts =[]
    counts =[]
    for b in blocks:
        for a in addends:
            rts.append(sum(data[(b,a)]) / len(data[(b,a)]))
            counts.append(len(data[(b,a)]))

    if show:
        print_analysis(rts,counts,blocks,addends,[totals[i] for i in addends])

    return (rts, counts)
```

The print_analysis function takes five parameters that describe a set of data for the task. The data is a list of response times and a list of corresponding correct answers. Then there are two lists that indicate the blocks and addend conditions represented in the data and finally a list of the total number of correct trials in each addend condition over all blocks. Those data are then displayed in a table.

```
def print_analysis(rts,counts,blocks,addends,totals):

    print()
    print("         ", end="")
    for a,t in zip(addends,map(lambda x: x/len(blocks), totals)):
        print("%6s (%2d) " % (a, t),end="")
    print()
    for b in range(len(blocks)):
        print("Block  %d" % blocks[b],end="")
        for a in range(len(addends)):
            print(" %6.3f (%2d)" %
                 (rts[a+b*len(addends)],counts[a+b*len(addends)]),end="")
        print()
```

The data_set variable is created to hold lists that represent all the problems to present in one set of the task.

```
data_set = [["a","2","c","k"],["d","2","f","k"],
            ["b","3","e","k"],["e","3","h","k"],
            ["c","4","g","k"],["f","4","j","k"],
            ["a","2","d","d"],["d","2","g","d"],
            ["b","3","f","d"],["e","3","i","d"],
            ["c","4","h","d"],["f","4","k","d"],
            ["a","2","c","k"],["d","2","f","k"],
            ["b","3","e","k"],["e","3","h","k"],
            ["c","4","g","k"],["f","4","j","k"],
            ["a","2","d","d"],["d","2","g","d"],
            ["b","3","f","d"],["e","3","i","d"],
            ["c","4","h","d"],["f","4","k","d"]]
```

The create_set function takes two parameters. The first indicates a number to specify which block of the experiment is being performed and the second indicates whether or not the trials should be shown in a visible window. It returns a randomly ordered list of 24 trial objects that make up one set of the control condition of the experiment.

```
def create_set(block,visible):

          return list(map(lambda x: trial(block,*x,visible=visible), actr.permute_list
(data_set)))
```

AGI command defaults

Two of the commands which we have seen in previous units are used slightly differently in the zbrodoff experiment. Previously when clear-exp-window was used we passed it the window to clear, but here we did not. If there is only one window opened by the AGI then most of the commands will default to working with that window and it does not need to be provided. Thus, this experiment assumes that there is only one open window and doesn't pass one to the clear-exp-window command. Similarly, when using add-text-to-exp-window (and other similar functions for buttons and lines which will be used later in the tutorial) the first parameter can be specified as nil (Lisp) or None (Python) to indicate that the default window should be used instead of specifying one. If there are multiple windows open when a call is made that indicates using the default window it will result in a warning and nothing will happen.

Monitoring function notes

As discussed with the unit 2 code, functions that are called as monitors are evaluated in separate threads and may require additional protection on changes to items which are also accessed outside of that function. What wasn't mentioned at that time is that when ACT-R is running to generate those actions there is some protection because the actions performed in the model are not evaluated in parallel – the events are evaluated one at a time. Therefore, when the monitor for output-key is called because the model pressed a key, you do not need to worry about threading issues between the monitoring function and other functions which are called as events by ACT-R or the code which called run, but it is still a potential problem when a person is performing the task since the person could

press the key in parallel with any of the code which is running the task while the monitor is active.

However, there is another issue to consider, and that is whether any functions that are used in the monitoring function have threading issues. In particular, when running a model you need to be careful about the ACT-R commands that are used since even though ACT-R will only evaluate one event at a time ACT-R itself is still "running". Most of the ACT-R commands presented in the tutorial are safe to use while ACT-R is running, in particular all the AGI commands for creating and manipulating windows are safe, but commands which run ACT-R cannot be used while it is already running and the commands for resetting and reloading a model cannot be used while it is still running. If you are using other ACT-R commands which are not described in the tutorial you will want to verify with the reference manual that they are safe for use while ACT-R is running, and if you're using the Lisp version it is strongly recommended that you not call any undocumented ACT-R functions since you won't know if they are safe to use.

The :ncnar Parameter

As was mentioned in the main text there is a new parameter being set in the models for this unit - :ncnar (normalize chunk names after run). This parameter toggles whether or not the system cleans up the references to merged chunks' names. If the parameter is set to **t**, which is the default, then the system will ensure that every slot of a chunk in the model which has a chunk as the value references the "true name" of the chunk in the slot i.e. the name of the original chunk in DM with which any copies have been merged. That operation can make debugging easier for the modeler because all the slot values will be consistent with the chunks shown to be in DM. However, if a model generates a lot of chunks and/or it makes many calls to one of the ACT-R commands to "run" the model it can take time to maintain that consistency. Thus it can be beneficial to turn this parameter off by setting it to **nil** when model debugging is complete and one just wants to collect the results or when the real time needed to run a model is important. For the models in the tutorial, leaving it enabled will typically not result in much of a run time increase (the paired model is the worst performer in this respect running around 10% slower with it enabled whereas the zbrodoff model shows effectively no difference since it does not generate a lot of chunks in DM and only involves a single call to run the model), but for tasks with more chunks in DM and/or more calls to run ACT-R one may find the savings from turning it off to be more significant.