Unit4 Model 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. Instead, what will be described in this document is a different way of writing experiments for models than you have seen in the previous units.

So far you have seen what can be described as a "trial at a time" or iterative approach to the experiments. The experiments run by executing some setup code, running the model to completion on that trial, recording a result and then repeating that process for the next trial. That style is a commonly used approach, but it has some drawbacks which you may have encountered. For instance, when debugging the model using the stepper it is not possible to stop the experiment. To stop the experiment you have to break out of the execution of the experiment function – the stop button of the stepper only terminates the model's run of the current trial. 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.

An alternative way to write the experiments is with an 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. Calling run 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 ACT-R commands like proc-display or goal-focus. However, as was seen in the sperling task, arbitrary events can also be scheduled to execute at particular simulated times. One can use the events generated by the model (key presses, mouse clicks, etc) as well as explicitly scheduled events to have an experiment that runs "with" the model instead of "around" it. Such an experiment only needs to call the run function one time to run the whole experiment instead of once (or more) per trial.

Having the model running in an event-driven experiment allows for more interactive control of the task as a whole. The stepper's stop button will now stop the whole experiment. That allows one to see exactly what is happening without having to abort the experiment function. 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. It can also make the model writing easier because one doesn't have to make sure that the model "stops" when it should, just that it can respond to the events that occur. The model then completes the task as a continuous process instead of as a sequence of separate interactions.

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

First, we will look at the **paired** model:

```
(defvar *response* nil)
(defvar *response-time* nil)
(defvar *model-doing-task* 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))

(defun paired-task (size trials &optional who)

  (if (not (eq who 'human))
      (do-experiment-model size trials)
    (do-experiment-person size trials)))

(defun do-experiment-model (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible nil)))


    (setf *model-doing-task* t)
    (reset)

    (install-device window)

    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))

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

          (setf *response* nil)
          (setf *response-time* nil)
          (setf start-time (get-time))

          (proc-display)
          (run-full-time 5)

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

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

          (proc-display)
          (run-full-time 5))

        (push (list (/ score size) (and (> score 0) (/ time (* score 1000.0)))) result)))

    (reverse result)))
```

```
(defun do-experiment-person (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible t)))

    (setf *model-doing-task* nil)
    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))

          (clear-exp-window)
          (add-text-to-exp-window :text (car x) :x 150 :y 150 :width 50)
          (setf *response* nil)
          (setf *response-time* nil)

          (setf start-time (get-time nil))
          (while (< (- (get-time nil) start-time) 5000)
                 (allow-event-manager window))

          (when (equal (second x) *response*)
            (incf score 1.0)
            (incf time (/ (- *response-time* start-time) 1000.0)))

          (clear-exp-window)
          (add-text-to-exp-window :text (second x) :x 150 :y 150)
          (sleep 5.0))

        (push (list (/ score size) (and (> score 0) (/ time score))) result)))

    ;; return the list of scores
    (reverse result)))

(defun paired-experiment (n)
  (do ((count 1 (1+ count))
       (results (paired-task 20 8)
                (mapcar (lambda (lis1 lis2)
                          (list (+ (first lis1) (first lis2))
                                (+ (or (second lis1) 0) (or (second lis2) 0))))
                  results (paired-task 20 8))))
      ((equal count n)
       (output-data results n))))

(defun output-data (data n)
  (let ((probability (mapcar (lambda (x) (/ (first x) n)) data))
        (latency (mapcar (lambda (x) (/ (or (second x) 0) n)) data)))
    (print-results latency *paired-latencies* "Latency")
    (print-results probability *paired-probability* "Accuracy")))

(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))


(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response* (string-upcase (string key)))
  (setf *response-time* (get-time *model-doing-task*)))
```

First we define some global variables to hold the key pressed, the time of the response,

and an indication of who is doing the task (model or person) so that we can record the time appropriately.

```
(defvar *response* nil)
(defvar *response-time* nil)
(defvar *model-doing-task* nil)
```

Then we define a list of the stimuli to be presented in the task.

```
(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")))
```

The experimental data to compare to are defined in global variables.

```
(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 the symbol human to run a person through the task instead of the model.

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

Unlike previous experiment code, in this case we have defined separate functions to run a model and to run a person. This function calls the appropriate one to run the task based on whether it's the model or a person running as a participant.

```
  (if (not (eq who 'human))
      (do-experiment-model size trials)
    (do-experiment-person size trials)))
```

The do-experiment-model function takes two parameters which are the number of pairs to present in a trial and the number of trials to run. 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-model (size trials)
```

Start by defining a variable to hold the results and opening a window for the task (note that the window is virtual because :visible is specified as **nil**).

```
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible nil)))
```

Set the *model-doing-task* variable appropriately.

```
(setf *model-doing-task* t)
```

Reset the model and tell it which window to interact with.

```
(reset)

(install-device window)
```

Repeat for the required number of trials.

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

Declare some local variables to hold the score and timing information.

```
  (let ((score 0.0)
        (time 0.0)
        (start-time))
```

Iterate over a randomized list of the required number of stimuli from the global set.

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

Clear the window and present the word from the pair.

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

Clear the response variables and record the trial start time.

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

Have the model process the display and run for exactly 5 seconds.

```
      (proc-display)
      (run-full-time 5)
```

If the answer provided was correct increment the score and the cumulative response time.

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

Clear the window and display the correct answer.

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

Have the model process the display and run for another 5 seconds.

```
          (proc-display)
          (run-full-time 5))
```

Compute the response results for the trial and save it on the list of results.

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

Return the list of results in the proper order.

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

The do-experiment-person function operates just like the do-experiment-model function except that it waits for a response from a person instead of running the model and it waits 5 seconds of real time during each presentation. For the number presentation that waiting is done using the Lisp command sleep (which will be described in detail below).

```
(defun do-experiment-person (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible t)))

    (setf *model-doing-task* nil)
    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))

          (clear-exp-window)
          (add-text-to-exp-window :text (car x) :x 150 :y 150 :width 50)
          (setf *response* nil)
          (setf *response-time* nil)

          (setf start-time (get-time nil))
          (while (< (- (get-time nil) start-time) 5000)
                 (allow-event-manager window))

          (when (equal (second x) *response*)
            (incf score 1.0)
            (incf time (/ (- *response-time* start-time) 1000.0)))

          (clear-exp-window)
          (add-text-to-exp-window :text (second x) :x 150 :y 150)
          (sleep 5.0))

        (push (list (/ score size) (and (> score 0) (/ time score))) result)))

    ;; return the list of scores
    (reverse result)))
```

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

```
(defun paired-experiment (n)
  (do ((count 1 (1+ count))
       (results (paired-task 20 8)
```

```
              (mapcar (lambda (lis1 lis2)
                        (list (+ (first lis1) (first lis2))
                              (+ (or (second lis1) 0) (or (second lis2) 0))))
                results (paired-task 20 8))))
     ((equal count n)
      (output-data results 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)
  (let ((probability (mapcar (lambda (x) (/ (first x) n)) data))
        (latency (mapcar (lambda (x) (/ (or (second x) 0) n)) data)))
    (print-results latency *paired-latencies* "Latency")
    (print-results probability *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))
```

The key handler for the window just records the key press and the time of that press in the global variables relying on the setting of *model-doing-task* to get the appropriate time.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response* (string-upcase (string key)))
  (setf *response-time* (get-time *model-doing-task*)))
```

The new functions used are run-full-time and sleep.

Run-full-time – this function takes one required parameter which is the time to run a model in seconds and a keyword parameter called :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 keyword parameter :real-time is specified as **t**, 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.

Sleep – sleep is a function defined in ANSI Common Lisp, but because it is being used for experiment generation it seems appropriate to discuss it. The Lisp specification for sleep says it takes one parameter, *seconds*, which is a non-negative real, and it causes execution to cease and become dormant for approximately the seconds of real time indicated by *seconds*, whereupon execution is resumed. An important thing to note is that this is only

useful when having a person do a task. **The sleep function will have no effect upon the timing of actions from the model's perspective**, but it will increase the time it takes to run the model from the user's perspective.

The other model for this unit is the **Zbrodoff** task and it is written using the event-driven approach. Because this is a more complex experiment than the previous tasks in the tutorial, there are some slightly more advanced Lisp facilities used to help keep things clearer and easier to use. Here is the code from the **zbrodoff** model:

```
(defvar *trials*)
(defvar *results*)
(defvar *start-time*)
(defvar *block*)

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

(defparameter *run-model* t)

(defstruct trial block addend1 addend2 sum answer visible)
(defstruct response block addend correct time)

(defun present-trial (trial &optional (new-window t))
  (let ((window (if new-window
                    (open-exp-window "Alpha-arithmetic Experiment"
                                     :visible (trial-visible trial))
                  nil)))

    (unless new-window
      (clear-exp-window))

    (add-text-to-exp-window :text (trial-addend1 trial) :x 100 :y 150 :width 25)
    (add-text-to-exp-window :text "+" :x 125 :y 150 :width 25)
    (add-text-to-exp-window :text (trial-addend2 trial) :x 150 :y 150 :width 25)
    (add-text-to-exp-window :text "=" :x 175 :y 150 :width 25)
    (add-text-to-exp-window :text (trial-sum trial) :x 200 :y 150 :width 25)

    (when new-window
      (install-device window))

    (proc-display :clear t)

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

    window))


(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (let ((trial (pop *trials*)))
    (push (make-response :block (trial-block trial)
                         :addend (trial-addend2 trial)
                         :time (/ (- (get-time *run-model*) *start-time*)
                                  1000.0)
                         :correct (string-equal (trial-answer trial)
                                                (string key)))
          *results*)
    (when *trials*
      (present-trial (first *trials*) nil))))
```

```
(defun collect-responses (trial-count)
  (setf *results* nil)
  (let ((window (present-trial (first *trials*))))
    (if *run-model*
        (run (* 10 trial-count))
      (while (< (length *results*) trial-count)
        (allow-event-manager window)))))



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

  (setf *block* 1)
  (setf *trials* (list (construct-trial (list addend1 addend2 sum answer)
                                        visible)))
  (collect-responses 1)
  (analyze-results))


(defun zbrodoff-set (&optional (visible (not *run-model*)))

  (setf *block* 1)
  (setf *trials* (create-set visible))
  (collect-responses 24)
  (analyze-results))


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

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

(defun zbrodoff (n)
  (let ((results nil))
    (dotimes (i n)
      (push (zbrodoff-experiment nil nil) results))

    (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)))))

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

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

```
(defun analyze-results (&optional (display t))
  (let ((blocks (sort (remove-duplicates (mapcar 'response-block *results*))
                      '<))
        (addends (sort (remove-duplicates (mapcar 'response-addend *results*)
                                          :test 'string-equal)
                       'string<))
        (counts nil)
        (rts nil)
        (total-counts nil))

    (setf total-counts (mapcar (lambda (x)
                                 (/ (count x *results*
                                           :key 'response-addend
                                           :test 'string=)
                                    (length blocks)))
                               addends))

    (dolist (x blocks)
      (dolist (y addends)
        (let ((data (mapcar 'response-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 display
      (print-analysis (reverse rts) (reverse counts) blocks addends
                      total-counts))

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


(defun print-analysis (rts counts blocks addends total-counts)
  (format t "~%        ")
  (dotimes (addend (length addends))
    (format t " ~6@a (~2d)" (nth addend addends) (nth addend total-counts)))
  (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))


(defun create-set (visible)
  (permute-list (mapcar (lambda (x) (construct-trial x visible))
                  '(("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")))))

(defun construct-trial (trial visible)
  (make-trial :block *block*
              :addend1 (first trial)
              :addend2 (second trial)
              :sum (third trial)
              :answer (fourth trial)
              :visible visible))
```

It starts by defining some global variables to hold the trials to present, the results collected, the start time of a trial, and a block count. This already shows some difference from the iterative approach because instead of storing the responses in variables which are collected by the "do-trial" type function the results are stored in a global variable for later use. Similarly, the trials are also stored in a global variable instead of being iterated over within a function.

```
(defvar *trials*)
(defvar *results*)
(defvar *start-time*)
(defvar *block*)
```

Then define the global variable that holds the experimental 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 there is a global variable defined which can be set 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** then 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 some custom structures to hold that data in a more descriptive format. A trial consists of a block number, the two addends to present, a sum to present, the correct answer, and whether or not to display the trial in a real window. A response contains the block in which it was given, the numeric addend of the trial (2, 3, or 4), whether the response was correct, and the response time.

```
(defstruct trial block addend1 addend2 sum answer visible)
(defstruct response block addend correct time)
```

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 (it will run faster if it doesn't have to create a new window for each trial). It does the same thing whether it is a person or a model performing the task.

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

If a new window is needed then it opens one setting the visible status of the window based on the setting in the trial structure otherwise it just sets window to nil.

```
  (let ((window (if new-window
                    (open-exp-window "Alpha-arithmetic Experiment"
                                     :visible (trial-visible trial))
                  nil)))
```

If a new window is not being used then it clears the existing window before displaying the current trial information.

```
    (unless new-window
      (clear-exp-window))

    (add-text-to-exp-window :text (trial-addend1 trial) :x 100 :y 150 :width 25)
    (add-text-to-exp-window :text "+" :x 125 :y 150 :width 25)
    (add-text-to-exp-window :text (trial-addend2 trial) :x 150 :y 150 :width 25)
    (add-text-to-exp-window :text "=" :x 175 :y 150 :width 25)
    (add-text-to-exp-window :text (trial-sum trial) :x 200 :y 150 :width 25)
```

Then if there is a new window it sets that as the device for the model to look at and then has the model processes it (if a person is doing the task these have no effect).

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

    (proc-display :clear t)
```

The time the trial starts is recorded in a global variable and the window value is returned.

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

    window))
```

The key handler for an event-driven experiment typically does more than just set variables for timing and response information. That's because it is the events, like keypresses, which will advance the task while the system runs instead of a function performing those actions while the system is not running. In this task, it starts by removing the current trial from the global list of trials.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (let ((trial (pop *trials*)))
```

Then, a structure that encodes the current response is added to the global list of responses.

```
    (push (make-response :block (trial-block trial)
                         :addend (trial-addend2 trial)
                         :time (/ (- (get-time *run-model*) *start-time*)
                                  1000.0)
                         :correct (string-equal (trial-answer trial)
                                                (string key)))
          *results*)
```

If there are more trials to present, then the next one is presented indicating that a new window should not be used. This is the critical difference between the iterative and event-based experiments. In the event-based experiment the participant's response leads directly to the presentation of the next trial. There is no need to stop the run and record a response before then running things again.

```
    (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 (trial-count)
```

The global set of results is cleared.

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

The first trial is presented and it records the window that was created.

```
  (let ((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 within 10 seconds or less per trial on average).

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

If a person is performing the task then the system waits for the appropriate number of responses to be recorded.

```
      (while (< (length *results*) trial-count)
        (allow-event-manager window)))))
```

The zbrodoff-trial 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. Thus, if the model is doing the task visible is **nil** in which case the window will be kept 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-trial (addend1 addend2 sum answer
                       &optional (visible (not *run-model*)))
```

Set the global values to indicate which block is being presented.

```
  (setf *block* 1)
```

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

```
  (setf *trials* (list (construct-trial (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 as described above for zbrodoff-trial. It runs once through a random permutation of the set of equations. 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 *block* 1)
  (setf *trials* (create-set visible))
  (collect-responses 24)
  (analyze-results))
```

The zbrodoff-block function takes one optional parameter as described above for do-trial. 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 *block* 1)
  (setf *trials* nil)
  (dotimes (i 8)
    (setf *trials* (append *trials* (create-set 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-results t))
  (reset)
  (setf *trials* nil)
  (dotimes (j 3)
    (setf *block* (+ j 1))
    (dotimes (i 8)
      (setf *trials* (append *trials* (create-set visible)))))
  (collect-responses 576)
  (analyze-results show-results))
```

The zbrodoff 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 (n)
  (let ((results nil))
```

Run the experiment n times with a virtual window and without displaying the individual analysis of each run.

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

Compute the average of the response times and 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 data table is displayed. 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.

```
(defun analyze-results (&optional (display t))
  (let ((blocks (sort (remove-duplicates (mapcar 'response-block *results*))
                      '<))
        (addends (sort (remove-duplicates (mapcar 'response-addend *results*)
                                          :test 'string-equal)
                       'string<))
        (counts nil)
        (rts nil)
        (total-counts nil))

    (setf total-counts (mapcar (lambda (x)
                                 (/ (count x *results*
                                           :key 'response-addend
                                           :test 'string=)
                                    (length blocks)))
                         addends))

    (dolist (x blocks)
      (dolist (y addends)
        (let ((data (mapcar 'response-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 display
      (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 total-counts)
  (format t "~%        ")
  (dotimes (addend (length addends))
    (format t " ~6@a (~2d)" (nth addend addends) (nth addend total-counts)))
  (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 create-set function takes one parameter which 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 (visible)
  (permute-list (mapcar (lambda (x) (construct-trial x visible))
                  '(("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 construct-trial function takes two parameters which are a list of 4 strings that represent a trial in the task and whether to display the trial in a visible window or not. The first three strings are the elements of the equation to display and the fourth is the key that should be pressed for a correct response. It returns a trial structure with the appropriate slot values set.

```
(defun construct-trial (trial visible)
```

```
(make-trial :block *block*
            :addend1 (first trial)
            :addend2 (second trial)
            :sum (third trial)
            :answer (fourth trial)
            :visible visible))
```

One final note about the zbrodoff code. As with the unit 2 demo model, the rpm-window-key-event-handler method is performing actions which are possibly unsafe for interfaces other than the ACT-R Environment windows and virtual windows to keep the code easier to understand. Details on alternatives can be found in the device-interaction-issues document in the docs directory.

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 chunk's 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 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 run for data generation or when the time to run a model really matters. For the models in the tutorial, leaving it enabled will not result in much of a run time increase (the paired model is the worst performer in this respect running around 15% slower with it enabled), but for tasks with more chunks or longer run times one may find the savings from turning it off to be more significant.

Compiling a model file

If a model is performing a task which involves a lot of Lisp code to present an experiment and collect data then compiling the model file to compile that Lisp code may decrease the time it takes to run the task. If you are using a Lisp with a GUI then one of the options on the file menu should be something similar to "Compile and Load". Using that to load the model will result in the file being compiled. It is also possible to explicitly compile the file before loading it using the Lisp compile-file function, and you will have to consult the Lisp documentation for details on how to use that. There is also an ACT-R command called compile-and-load which can be used. That command takes one parameter which must be the name of a file which has a name ending in ".lisp". It will compile that file and then load that compiled version of the code. The "Load Model" button in the ACT-R Environment's Control Panel will not compile files by default, but it is possible to enable that functionality and you can consult the Environment's manual for details.

When compiling a model file one important thing to be aware of is that if you change the model file you will want to compile it again when you load it after you save the changes. You should also be careful if using the ACT-R reload command or the "Reload" button on the Environment's Control Panel when working with a compiled model file. The reload command in ACT-R loads the last file which contained a call to the ACT-R clear-all command at the top level. If that was a compiled file then it will load that same compiled file again which will not reflect any changes that have occurred in the original model file.