

The experiments for this unit are a little more complex than the previous ones, but they do not use any new experiment generation functions. The fan experiment includes an alternate version of the model running function (it is commented out of the experiment and thus not used) that will be discussed. The alternate function runs the model through the task without using the visual interface for input or the motor module for output. Instead it puts the input into goal slots before running the model and reads the response from a goal slot upon completion. This is done through the use of two functions that manipulate chunks outside of the model and such a mechanism can be used when the details of the visual/motor systems are not of interest or speed of running the model is really important (the assignment model works in such a fashion).

Here is the experiment code for the **fan model**. This experiment differs from most that have been presented so far in that it only presents the test portion of the task. The study portion has been encoded in the model explicitly and to do it as a person you need to first study the required items. For the model, the experiment will favor one of the two provided productions during two iterations over the task (once favoring one and then the other) so that the results can be averaged using only 2 runs. This is another case of the experiment being simplified for the tutorial. Another way to accomplish this would be to allow those two productions to compete equally and thus they would each fire on average half the time. Then the model would be run for many iterations over the task and the results would be averaged to generate the predictions, but because there is no noise in the model and the only source of differences are foil trials this simplification is sufficient.

Here are the contents of the **Misc window**:

```
(defconstant *person-location-data* '(1.11 1.17 1.22
                                     1.17 1.20 1.22
                                     1.15 1.23 1.36
                                     1.20 1.22 1.26
                                     1.25 1.36 1.29
                                     1.26 1.47 1.47))

(defvar *response*)
(defvar *response-time*)

(defun test-sentence-model (person location target term)
  (let ((window (open-exp-window "Sentence Experiment"
                                :visible t
                                :width 600
                                :height 300))
        (x 25))

    (reset)
    (pm-install-device window)

    (case term
      (person (spp retrieve-from-person :c 0))
      (location (spp retrieve-from-location :c 0)))

    (dolist (text (list "The" person "is" "in" "the" location))
      (add-text-to-exp-window :text text :x x :y 150 :width 75)
      (incf x 75))

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

```

(pm-proc-display)

(pm-run 30.0)

(if (null *response*)
    (list 30.0 nil)
    (list *response-time*
          (or (and target (string-equal *response* "k"))
              (and (null target) (string-equal *response* "d"))))))

#| This version of the test-sentence-model function
   runs the model without using the visual interface.
   It is provided as an example of showing how one could
   bypass the interface when it isn't really necessary.
   The difference is explained in the experiment description
   text.

   The assignment for the unit MUST work with the original version
   NOT this one.

(defun test-sentence-model (person location target term)

  (reset)

  (case term
    (person (spp retrieve-from-person :c 0))
    (location (spp retrieve-from-location :c 0)))

  (mod-chunk-fct 'goal (list 'arg1 person 'arg2 location))

  (setf *response-time* (pm-run 30.0))

  (setf *response* (chunk-slot-value goal state))

  (list *response-time*
        (or (and target (string-equal *response* "k"))
            (and (null target) (string-equal *response* "d")))))

|#

(defun test-sentence-person (person location target term)
  (let ((window (open-exp-window "Sentence Experiment" :width 600 :height 300))
        (x 25)
        (start-time))

    (dolist (text (list "The" person "is" "in" "the" location))
      (add-text-to-exp-window :text text :x x :y 150)
      (incf x 75))

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

    (setf start-time (/ (pm-get-time) 1000.0))

    (while (null *response*) ;; wait for a key to be pressed by a person
      (allow-event-manager window))

    (list (- *response-time* start-time)
          (or (and target (string-equal *response* "k"))
              (and (null target) (string-equal *response* "d")))))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response-time* (/ (pm-get-time) 1000.0))
  (setf *response* (string key)))

```

```

(defun do-person-location (term &optional (in-order *actr-enabled-p*))
  (let ((test-set '("lawyer" "store" t)("captain" "cave" t)
        ("hippie" "church" t)("debutante" "bank" t)
        ("earl" "castle" t)("hippie" "bank" t)
        ("fireman" "park" t)("captain" "park" t)
        ("hippie" "park" t) ("fireman" "store" nil)
        ("captain" "store" nil)("giant" "store" nil)
        ("fireman" "bank" nil)("captain" "bank" nil)
        ("giant" "bank" nil)("lawyer" "park" nil)
        ("earl" "park" nil)("giant" "park" nil)))
    (results nil))

  (dolist (sentence (if in-order test-set (permute-list test-set)))
    (push (list sentence
                (apply (if *actr-enabled-p*
                          #'test-sentence-model
                          #'test-sentence-person)
                      (append sentence (list term))))
          results))

  (mapcar #'second (sort results #'< :key #'(lambda (x)
                                             (position (car x) test-set))))))

(defun average-person-location ()
  (output-person-location (mapcar #'(lambda (x y)
                                     (list (/ (+ (car x) (car y)) 2.0)
                                           (and (cadr x) (cadr y))))
                            (do-person-location 'person)
                            (do-person-location 'location))))

(defun output-person-location (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *person-location-data*)
    (mean-deviation rts *person-location-data*)
    (format t "~%TARGETS:~%                Person fan~%"
            (format t "  Location          1                2                3~%"
                    (format t "    fan")
                    (format t "    fan")
                    (format t "    fan"))
            (dotimes (i 3)
              (format t "~%          ~d          " (1+ i))
              (dotimes (j 3)
                (format t "~{~8,3F (~3s)~}" (nth (+ j (* i 3)) data))))
            (format t "~%~%FOILS:"
                    (dotimes (i 3)
                      (format t "~%          ~d          " (1+ i))
                      (dotimes (j 3)
                        (format t "~{~8,3F (~3s)~}" (nth (+ j (* (+ i 3) 3)) data)))))))


```

Here are the contents of the **commands window**:

```

(sgp :v t :esc t :lf 0.64 :mas 3)

(setf *actr-enabled-p* t)

(set-general-base-levels
 (guard* 10) (beach* 10) (castle* 10) (dungeon* 10) (earl* 10)
 (forest* 10) (hippie* 10) (park* 10) (church* 10) (bank* 10)
 (captain* 10) (cave* 10) (giant* 10) (debutante* 10) (fireman* 10)
 (lawyer* 10) (store* 10) (in* 10))

(goal-focus goal)

```

The only thing new there is the call to **set-general-base-levels**. It is used to set the base level activation of those chunks (the B_i in the activation equation) to a "large" value so that they will not fail to be retrieved and their retrieval times will be essentially zero.

Now here are the definitions from the **Misc window**. First it defines a global constant that holds the experimental data that the model is to be fit to.

```
(defconstant *person-location-data* '(1.11 1.17 1.22
                                     1.17 1.20 1.22
                                     1.15 1.23 1.36
                                     1.20 1.22 1.26
                                     1.25 1.36 1.29
                                     1.26 1.47 1.47))
```

Then it defines the global variables that will hold the response and the response time.

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

The `test-sentence-model` function takes 4 parameters. The first, `person`, is the string of the person to display in the sentence. The second, `location`, is the string of the location to display. The third, `target`, is to be either `t` or `nil` to indicate whether this is a target or a foil trial respectively, and the fourth, `term`, specifies which of the productions to favor in retrieving the study item and should be either the symbol `person` or `location`. The function returns a list of two items. The first is the response time of the key press in seconds or 30.0 if there was no response and the second item is `t` if the response was correct (`k` for a target and `d` for a foil) or `nil` if there was no response or the response was incorrect.

```
(defun test-sentence-model (person location target term)
```

First open a window.

```
(let ((window (open-exp-window "Sentence Experiment"
                              :visible t
                              :width 600
                              :height 300)))
```

Create a variable to hold the x coordinates of the items to present.

```
(x 25))
```

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

```
(reset)
(pm-install-device window)
```

Set the cost of the requested production to 0 so that it will be favored.

```
(case term
  (person (spp retrieve-from-person :c 0))
```

```
(location (spp retrieve-from-location :c 0)))
```

Iterate over the words of the sentence and display each one 75 pixels from the previous one.

```
(dolist (text (list "The" person "is" "in" "the" location))
  (add-text-to-exp-window :text text :x x :y 150 :width 75)
  (incf x 75))
```

Clear the response variables.

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

Make the model process the display and run for up to 30 seconds.

```
(pm-proc-display)
(pm-run 30.0)
```

If there was no response return the list of 30.0 and nil otherwise return the list of the response time in seconds and whether the response was correct or incorrect.

```
(if (null *response*)
    (list 30.0 nil)
    (list *response-time*
          (or (and target (string-equal *response* "k"))
              (and (null target) (string-equal *response* "d"))))))
```

Next is a block of comments (the #| and|# are the start and end markers for a block of comments in Lisp) which contain a description of a function and its definition. As indicated in the description this function is a replacement for test-sentence-model which was described above. The version in the comments operates without using a display for interacting.

Using the replacement function will result in generating a somewhat different model than the one that works with the default function because it will not use the visual or motor modules, but the predictions can be made identical. The model is probably easier to generate and will run faster without the overhead of generating a display, but does require extra work to estimate the “action” times since they will not be automatically produced by the respective modules.

Depending on the objectives of the modeling sometimes the actions aren't really important and using a simplified representation or direct manipulation of the chunks is sufficient. Prior to ACT-R 5, many models operated in such a manner.

```
#| This version of the test-sentence-model function
    runs the model without using the visual interface.
    It is provided as an example of showing how one could
    bypass the interface when it isn't really necessary.
    The difference is explained in the experiment description
    text.
```

The assignment for the unit MUST work with the original version
NOT this one.

```
(defun test-sentence-model (person location target term)
```

First, reset the model and set the production parameters to prefer the required retrieval production.

```
(reset)

(case term
  (person (spp retrieve-from-person :c 0))
  (location (spp retrieve-from-location :c 0)))
```

Next, modify the chunk named goal placing the strings directly into the slots. The mod-chunk-fct function will be discussed in more detail below.

```
(mod-chunk-fct 'goal (list 'arg1 person 'arg2 location))
```

Set the response time variable to the value returned by running the model.

```
(setf *response-time* (pm-run 30.0))
```

Set the response to the value in the state slot of the chunk named goal. The chunk-slot-value function (actually a macro) will be described below.

```
(setf *response* (chunk-slot-value goal state))
```

Return the list of response time and correctness as above.

```
(list *response-time*
      (or (and target (string-equal *response* "k"))
          (and (null target) (string-equal *response* "d")))))
|#
```

The test-sentence-person function operates like the test-sentence-model function except that the term parameter is ignored and it waits for a person to press a key instead of the model.

```
(defun test-sentence-person (person location target term)
  (let ((window (open-exp-window "Sentence Experiment" :width 600 :height 300))
        (x 25)
        (start-time))

    (dolist (text (list "The" person "is" "in" "the" location))
      (add-text-to-exp-window :text text :x x :y 150)
      (incf x 75))

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

    (setf start-time (/ (pm-get-time) 1000.0))

    (while (null *response*) ;; wait for a key to be pressed by a person
      (allow-event-manager window))

    (list (- *response-time* start-time)
          (or (and target (string-equal *response* "k"))
              (and (null target) (string-equal *response* "d")))))
```

The key handler for the window just records the key press and the time of that press (in seconds) in the global variables.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response-time* (/ (pm-get-time) 1000.0))
  (setf *response* (string key)))
```

The do-person-location function takes one required parameter, term, which specifies which production to favor for the model as specified for test-sentence-model and one optional parameter, in-order, which defaults to the value of the global variable *actr-enabled-p*. It iterates over all of the test sentences for the task collecting the data as returned by the test-sentence-model or test-sentence-person function. If in-order is t the items are presented in the default order and if it is nil they are presented in a random order. It returns a list of responses in the same order as the results in the global data list.

```
(defun do-person-location (term &optional (in-order *actr-enabled-p*))
  (let ((test-set '(("lawyer" "store" t)("captain" "cave" t)
                  ("hippie" "church" t)("debutante" "bank" t)
                  ("earl" "castle" t)("hippie" "bank" t)
                  ("fireman" "park" t)("captain" "park" t)
                  ("hippie" "park" t) ("fireman" "store" nil)
                  ("captain" "store" nil)("giant" "store" nil)
                  ("fireman" "bank" nil)("captain" "bank" nil)
                  ("giant" "bank" nil)("lawyer" "park" nil)
                  ("earl" "park" nil)("giant" "park" nil)))
    (results nil))

  (dolist (sentence (if in-order test-set (permute-list test-set)))
    (push (list sentence
                (apply (if *actr-enabled-p*
                          #'test-sentence-model
                          #'test-sentence-person)
                      (append sentence (list term))))
          results))

  (mapcar #'second (sort results #'< :key #'(lambda (x)
                                             (position (car x) test-set))))))
```

The average-person-location function takes no parameters. It calls the do-person-location function once to favor the person and once to favor the location. It returns a list of responses that are the average of the two response times for a given condition and t if both responses were correct.

```
(defun average-person-location ()
  (output-person-location (mapcar #'(lambda (x y)
                                     (list (/ (+ (car x) (car y)) 2.0)
                                           (and (cadr x) (cadr y))))
                            (do-person-location 'person)
                            (do-person-location 'location))))
```

The output-person-location function takes one parameter which is a list of responses that are in the same order as the global experimental data. It prints the comparison of that

data to the experimental results and prints a table of the response times for targets and foils.

```
(defun output-person-location (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *person-location-data*)
    (mean-deviation rts *person-location-data*)
    (format t "~%TARGETS:~%"          Person fan~%)
    (format t "   Location      1      2      3~%")
    (format t "   fan")

    (dotimes (i 3)
      (format t "~%      ~d      " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* i 3)) data))))

    (format t "~%~%FOILS:")
    (dotimes (i 3)
      (format t "~%      ~d      " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* (+ i 3) 3)) data))))))
```

The two new functions (or macros) there are `mod-chunk-fct` and `chunk-slot-value`. These are described in the ACT-R manual, but I'll also discuss them here. I'm going to start with a discussion of the difference between functions and macros in Lisp and why ACT-R uses one or the other in certain places and the impact that it has (if you don't care to know that level of detail you can skip down to the descriptions of the new operators).

First, there is a distinction between functions and macros in Lisp. When calling a function the parameters are evaluated first, but a macro does not evaluate its parameters (that's not the only difference, but probably the most significant for current purposes). Many of the ACT-R functions you've seen are actually macros e.g. `chunk-type`, `add-dm`, and `p`. They are macros so that you don't have to worry about quoting symbols or lists and other issues with Lisp syntax, but because they don't evaluate their parameters there are some things that you can't do with the macros (at least not without using an `eval` and backquoting but I'm not going to discuss backquoting either). For example, say you have a variable called `*number*` and you'd like to create a chunk that has the value of `*number*` in one of its slots. The following is not going to work:

```
CG-USER(39): (defvar *number* 3)
*NUMBER*
CG-USER(40): (chunk-type test slot)
TEST
CG-USER(41): (add-dm (foo isa test slot *number*))
(FOO)
```

because the macro isn't going to evaluate the variable `*number*` to get its value but instead will create a chunk that literally contains `*number*` as shown here (`wm` is another ACT-R macro that prints out a chunk given its name):

```
CG-USER(42): (wm foo)
Foo      0.000
  isa TEST
  slot *Number*
```

To get around that issue most of the ACT-R macros have a function that does the same thing (in fact most of the macros just expand to a call to the function) and the naming convention is to add “-fct” to the name for the functional form. However, when using the function you have to pay more attention to Lisp syntax and make sure that symbols are quoted and lists are constructed appropriately, and often the required parameters are a little different – things that do not need to be in a list for the macro version need to be in a list for the function. For example, add-dm-fct requires a list of lists as parameters instead of just an arbitrary number of lists. Here is the code that would generate the chunk as desired above:

```
CG-USER(43): (add-dm-fct (list (list 'foo 'isa 'test 'slot *number*)))
(FOO)
CG-USER(44): (wm foo)
  Foo    0.000
    isa TEST
    slot 3
```

That’s not the only way to construct the lists (there are many ways one could accomplish the same thing) but should make the difference clear.

Now that I’ve covered that here are the new operators.

mod-chunk and **mod-chunk-fct** – This command modifies a chunk. The macro version requires a chunk name and then any number of slot and value pairs and the function requires a chunk name and then a list of slot and value pairs. Each of the slots specified for the chunk is given the corresponding value. It returns the name of the chunk which was modified. Here is an example of each being used on the chunk foo created above:

```
CG-USER(45): (mod-chunk foo slot 5)
FOO
CG-USER(46): (wm foo)
  Foo    0.000
    isa TEST
    slot 5
(FOO)

CG-USER(47): (mod-chunk-fct 'foo (list 'slot 6))
FOO
CG-USER(48): (wm foo)
  Foo    0.000
    isa TEST
    slot 6
(FOO)
```

chunk-slot-value and **chunk-slot-value-fct** – This command returns the value in a particular slot of a chunk. The parameters for both versions are the name of the chunk and the name of the slot. Here is an example of each again being used on the chunk foo:

```
CG-USER(49): (chunk-slot-value foo slot)
6

CG-USER(50): (chunk-slot-value-fct 'foo 'slot)
6
```

The **Grouped** model is really just a demonstration of partial matching. The experiment code is only there to collect and display key presses of the model. It is not an interactive experiment which a person can also do nor does it have a direct comparison to any existing data.

Here is the code from the **Misc Window**:

```
(defvar *response* nil)

(defun run-grouped-recall ()
  (setf *response* nil)
  (reset)
  (pm-run 20.0)
  (reverse *response*))

(defun record-response (value)
  (push value *response*))
```

and here is the code from the **Commands Window**:

```
(sgp :rt -.5 :ans .15 :ga 0 :esc t :pm t :v t :pmt t :act t)

(setf *actr-enabled-p* t)

(goal-focus goal)

(setsimilarities
 (first second -0.5)
 (second third -0.5)
 (first third -1))
```

Looking at the commands first we see basically all of the activation parameters being set:

```
(sgp :rt -.5 :ans .15 :ga 0 :esc t :pm t :v t :pmt t :act t)
```

The parameter `:pm`, partial matching, is set to `t` to enable that functionality, and `:pmt`, the partial matching trace, enables the printing of the details of the partial matching process during the run.

The next couple of lines are the same as have been seen many times before to set the model as the participant and set the initial contents of the goal buffer.

```
(setf *actr-enabled-p* t)

(goal-focus goal)
```

Then there is another new command for this unit, `setsimilarities`.

```
(setsimilarities
 (first second -0.5)
 (second third -0.5)
 (first third -1))
```

This command is used to set the similarity between pairs of chunks in ACT-R. Those are the M_{ki} values from the activation equation. The parameters to `setsimilarities` (the macro form of the command) are an arbitrary number of three element lists. The first two elements of a list must name chunks and the third value is the M_{ki} between them, which is set symmetrically.

The experiment code in the Misc window is pretty simple. First it defines a global variable `*response*` that will be used to hold the model's responses.

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

The `run-grouped-recall` function takes no parameters. It clears the `*response*` list, runs the model, and then returns the list of responses in order. Because the responses are pushed onto the front of the list by the data collection function below it needs to be reversed before returning it.

```
(defun run-grouped-recall ()
  (setf *response* nil)
  (reset)
  (pm-run 20.0)
  (reverse *response*))
```

The data collection is not done through pressing keys because there isn't an interface to accept them. Instead, the model simulates that process by directly calling a Lisp function to record the response. How that is done will be described next. This is the function that is called, and all it does is push the response onto the global list.

```
(defun record-response (value)
  (push value *response*))
```

Calling Lisp code from within a production is something new in this model, and it's done in the `harvest-first-item`, `harvest-second-item`, and `harvest-third-item` productions. Here is the `harvest-third-item` production.

```
(p harvest-third-item
  =goal>
  isa      recall-list
  step     recalling-item
  element  third
  group    =group
  =retrieval>
  isa      item
  name     =name
==>
  =retrieval>
  status   retrieved
  =goal>
  element  fourth
  +retrieval>
  isa      item
  parent   =group
  position fourth
  - status retrieved

  !eval! (record-response =name)
)
```

The new operation shown in that production is !eval!. [The ‘!’ is called bang in Lisp, so that’s pronounced bang-eval-bang.] It can be placed on either the LHS or RHS of a production and must be followed by a Lisp expression which will be evaluated.

On the RHS of a production all it does is evaluate an expression. When the production fires, the !eval! is just another action that occurs. Note that the expression can contain variables from the production and the current binding of that variable in the instantiation is what will be used in the expression.

However, on the LHS of a production a !eval! actually specifies a condition that must be met before the production can be selected as with all other LHS conditions. The value returned by the evaluation of a LHS !eval! call must be non-nil for the production to be selected. Because it will be called during the selection process, a LHS !eval! is going to be evaluated very often and even when the production that it’s in is not the one that will be eventually selected and fired.

!eval! is a powerful tool, but one that can easily be abused. Using it to call Lisp as an abstraction for an aspect of the model which isn’t necessary to model in detail for a particular task is the main recommended use. In general, the predictions of the model shouldn’t depend heavily on the use of !eval!, otherwise there isn’t really any point to using ACT-R to model – you might as well just generate some functions to produce the data you want.

In this model it is used to collect the model’s responses without needing the overhead of creating an experiment with which to interact. Because this task isn’t concerned with response time or stimuli acquisition there isn’t really a need for an interactive experiment because it would only serve to increase the size and complexity of the model without adding anything to its purpose.

The assignment is the **Siegler** model which contains a “model only” experiment because there is no display with which to interact. However, there aren’t any functions used which haven’t been explained so it should be easy to follow.

Here are the contents of the **Misc Window**:

```
(defvar *response*)

(defconstant *siegler-data* '((0 .05 .86 0 .02 0 .02 0 0 .06)
                             (0 .04 .07 .75 .04 0 .02 0 0 .09)
                             (0 .02 0 .10 .75 .05 .01 .03 0 .06)
                             (.02 0 .04 .05 .80 .04 0 .05 0 0)
                             (0 0 .07 .09 .25 .45 .08 .01 .01 .06)
                             (.04 0 0 .05 .21 .09 .48 0 .02 .11))

  "The experimental data to be fit")

(defun test-fact (arg1 arg2)
  (reset)
  (mod-chunk-fct 'goal (list 'arg1 arg1 'arg2 arg2)))
```

```

(pm-run 30.0)
(chunk-slot-value goal answer))

(defun do-one-set ()
  (list (test-fact "one" "one")
        (test-fact "one" "two")
        (test-fact "one" "three")
        (test-fact "two" "two")
        (test-fact "two" "three")
        (test-fact "three" "three")))

(defun run-subjects (n)
  (let ((responses nil))
    (dotimes (i n)
      (push (do-one-set) responses))
    (analyze responses)))

(defun analyze (responses)
  (let ((results (mapcar #'(lambda (x)
                            (mapcar #'(lambda (y)
                                        (/ y (length responses))) x))
                          (apply #'mapcar #'(lambda (&rest z)
                                              (let ((res nil))
                                                (dolist (i '("zero" "one" "two" "three"
"four" "five" "six" "seven" "eight") res)
          (push (count i z :test #'string-equal) res)
          (setf z (remove i z :test #'string-equal)))
          (push (length z) res)
          (reverse res)))
                                              responses))))))
    (display-results results)))

(defun display-results (results)
  (let ((questions '("1+1" "1+2" "1+3" "2+2" "2+3" "3+3")))
    (correlation results *siegler-data*)
    (mean-deviation results *siegler-data*)
    (format t "      0      1      2      3      4      5      6      7      8
Other~%"
            (dotimes (i 6)
              (format t "~a~{~6,2f~}~%" (nth i questions) (nth i results))))))

```

and here is the code from the **Commands Window**:

```

(sgp :rt 0.0 :esc t :v nil :ga 0 :pm t :pmt nil :ans .1)

(set-general-base-levels
 (zero 10) (one 10) (two 10) (three 10) (four 10) (five 10)
 (six 10) (seven 10) (eight 10) (nine 10))

(setf *actr-enabled-p* t)

(goal-focus goal)

```

Everything in the **Commands Window** has also been seen before, so there is no need to go into detail about it and we will go right into the **Misc Window** code.

Start by defining a global variable to hold the response.

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

Then define a global constant that holds the experimental data to be fit.

```
(defconstant *sieglar-data* '((0 .05 .86 0 .02 0 .02 0 0 .06)
                              (0 .04 .07 .75 .04 0 .02 0 0 .09)
                              (0 .02 0 .10 .75 .05 .01 .03 0 .06)
                              (.02 0 .04 .05 .80 .04 0 .05 0 0)
                              (0 0 .07 .09 .25 .45 .08 .01 .01 .06)
                              (.04 0 0 .05 .21 .09 .48 0 .02 .11))
  "The experimental data to be fit")
```

The test-fact function takes two parameters which are the text of the items to present.

```
(defun test-fact (arg1 arg2)
```

The model is reset.

```
(reset)
```

The goal chunk is modified to contain the text items.

```
(mod-chunk-fct 'goal (list 'arg1 arg1 'arg2 arg2))
```

Then the model is run to generate a response.

```
(pm-run 30.0)
```

The model's response is whatever is in the answer slot of the chunk named goal, and that value is returned.

```
(chunk-slot-value goal answer)
```

The do-one-set function takes no parameters. It runs one trial of each of the 6 stimuli and returns the list of the responses.

```
(defun do-one-set ()
  (list (test-fact "one" "one")
        (test-fact "one" "two")
        (test-fact "one" "three")
        (test-fact "two" "two")
        (test-fact "two" "three")
        (test-fact "three" "three")))
```

The run-subjects function takes one parameter which is the number of times to repeat the set of 6 test stimuli. It runs that many times over the set of stimuli and then outputs the results.

```
(defun run-subjects (n)
  (let ((responses nil))
    (dotimes (i n)
      (push (do-one-set) responses))
    (analyze responses)))
```

The analyze function takes one parameter which is a list of lists where each sublist contains the responses to the 6 test stimuli. It calculates the percentage of each of the

answers from zero to eight or other from the given trial data and calls display-results to print out that information.

```
(defun analyze (responses)
  (let ((results (mapcar #'(lambda (x)
                            (mapcar #'(lambda (y)
                                        (/ y (length responses))) x))
                          (apply #'mapcar #'(lambda (&rest z)
                                              (let ((res nil))
                                                (dolist (i '("zero" "one" "two" "three"
"four" "five" "six" "seven" "eight") res)
          (push (count i z :test #'string-equal) res)
          (setf z (remove i z :test #'string-equal)))
          (push (length z) res)
          (reverse res))))
                          responses))))
  (display-results results)))
```

The display-results function takes one parameter which is a list of six lists where each sublist is a list of nine items which represents the percentages of the answers 0-8 or other for each of the 6 test stimuli. It prints out the comparison to the experimental data and then displays the table of the results.

```
(defun display-results (results)
  (let ((questions '("1+1" "1+2" "1+3" "2+2" "2+3" "3+3")))
    (correlation results *sieglar-data*)
    (mean-deviation results *sieglar-data*)
    (format t "          0      1      2      3      4      5      6      7      8
Other~%"
            (dotimes (i 6)
              (format t "~a~{~6,2f~}~%" (nth i questions) (nth i results)))))
```