

## Unit7 Model Code Description

The paired associate model for this unit uses the same experiment code as the paired associate model from unit 4. So in this document we will only be looking at the past tense model. This is another example of an experiment and model that do not use the perceptual and motor components of ACT-R.

```
(defvar *report*)
(defvar *word*)
(defvar *repcount*)
(defvar *trial*)

(defun make-word-freq-list (l &optional (start 0))
  (when l
    (let ((count (third l)))
      (cons (list (+ start count) (first l) (fourth l)
                (if (eq (second l) 'I) 'blank 'ed))
            (make-word-freq-list (nthcdr 4 l) (+ start count))))))

(defparameter *word-list*
  (make-word-freq-list '(have      I      12458 had
                        do        I      4367 did
                        make      I      2312 made
                        get        I      1486 got
                        use        R      1016 use
                        look       R      910  look
                        seem       R      831  seem
                        tell       I      759  told
                        show       R      640  show
                        want       R      631  want
                        call       R      627  call
                        ask        R      612  ask
                        turn       R      566  turn
                        follow     R      540  follow
                        work       R      496  work
                        live       R      472  live
                        try        R      472  try
                        stand      I      468  stood
                        move       R      447  move
                        need       R      413  need
                        start      R      386  start
                        lose       I      274  lost)))

(defparameter *total-count* (caar (last *word-list*)))

(defun random-word ()
  (let ((num (act-r-random *total-count*)))
    (cdr (find-if (lambda (x) (< num (first x))) *word-list*)))

(defun make-one-goal ()
  (setf *word* (random-word))

  (set-buffer-chunk 'imaginal
    (car (define-chunks-fct
          (list (list 'verb (first *word*))))))
  (goal-focus starting-goal))

(defun add-past-tense-to-memory ()
```

```

(let ((word (random-word)))
  (set-buffer-chunk 'imaginal
    (car (define-chunks-fct
          (list (mapcan (lambda (x y) (list x y))
                        '(verb stem suffix word))))))
  (clear-buffer 'imaginal)))

(defun report-irreg (&optional (graph nil) (trials 1000))
  (format t "~% Irreg   Reg   None   Overreg~%"
    (let ((data (mapcar 'fourth (rep-f-i *report* trials))))
      (when (and graph data)
        (graph-it data)))
    nil))

(defun graph-it (data)
  (let* ((win (open-exp-window "Irregular Verbs correct" :width 500 :height 475))
        (low (apply 'min data))
        (zoom (min .9 (/ (floor low .1) 10))))
    (allow-event-manager win)
    (clear-exp-window)
    (add-text-to-exp-window :x 5 :y 5 :text "1.0" :width 22)
    (add-text-to-exp-window :x 5 :y 400 :text (format nil "~0,2f" zoom) :width 22)
    (add-text-to-exp-window :x 5 :y 200 :text (format nil "~0,2f"
                                                    (+ zoom (/ (- 1.0 zoom) 2)))
                            :width 22)

    (add-text-to-exp-window :x 200 :y 420 :text "Trials" :width 100)
    (add-line-to-exp-window '(30 10) '(30 410) :color 'black)
    (add-line-to-exp-window '(450 410) '(25 410) :color 'black)
    (dotimes (i 10)
      (add-line-to-exp-window (list 25 (+ (* i 40) 10))
                              (list 35 (+ (* i 40) 10))
                              :color 'black))

    (when (= (length data) 1)
      (push (first data) data))
    (do* ((increment (max 1.0 (floor (/ 450.0 (length data)))))
         (range (floor 400 (- 1.0 zoom)))
         (intercept (+ range 10))
         (p1 (butlast data) (cdr p1))
         (p2 (cdr data) (cdr p2))
         (last-x 30 this-x)
         (last-y (- intercept (floor (* range (car p1))))
                  (- intercept (floor (* range (car p1)))))
         (this-x (+ last-x increment)
                  (+ last-x increment))
         (this-y (- intercept (floor (* range (car p2))))
                  (- intercept (floor (* range (car p2)))))
         ((null (cdr p1)) (add-line-to-exp-window
                          (list last-x last-y) (list this-x this-y) :color 'red))
         (add-line-to-exp-window (list last-x last-y)
                                  (list this-x this-y)
                                  :color 'red))
      (allow-event-manager win)))

(defun rep-f-i (l n)
  (when l
    (let ((x (if (> (length l) n) (subseq l 0 n) l))
          (y (if (> (length l) n) (subseq l n) nil))
          (irreg 0)
          (reg 0))
      (let ((word (random-word)))
        (set-buffer-chunk 'imaginal
          (car (define-chunks-fct
                (list (mapcan (lambda (x y) (list x y))
                              '(verb stem suffix word))))))
        (clear-buffer 'imaginal)))
      (report-irreg graph trials)
      (graph-it data)
      (rep-f-i (cons x y) (1- n))))

```

```

        (none 0)
        (error 0))
(dolist (i x)
  (when (first i)
    (case (second i)
      (reg
       (incf reg))
      (irreg
       (incf irreg))
      (none
       (incf none))
      (t
       (incf error))))))

(let* ((total (+ irreg reg none))
      (data
       (if (zerop total)
           (list 0 0 0 0)
           (list (/ irreg total) (/ reg total)
                 (/ none total) (if (> (+ irreg reg) 0) (/ irreg (+ irreg reg)) 0))))))
  (format t "~{-6,3F-^ ~}~%" data)
  (cons data (rep-f-i y n))))))

(defun add-to-report (the-chunk)
  (let ((stem (chunk-slot-value-fct the-chunk 'stem))
        (word (chunk-slot-value-fct the-chunk 'verb))
        (suffix (chunk-slot-value-fct the-chunk 'suffix))
        (irreg (eq (third *word*) 'blank)))

    (if (eq (first *word*) word)
        (cond
         ((and (eq stem word) (eq suffix 'ed))
          (push-last (list irreg 'reg) *report*))
         ((and (null suffix) (null stem))
          (push-last (list irreg 'none) *report*))
         ((and (eq stem (second *word*)) (eq suffix 'blank))
          (push-last (list irreg 'irreg) *report*))
         (t
          (print-warning "Incorrectly formed verb. Presented ~s and produced ~{-s-^ ~}."
                        (first *word*)
                        (mapcan (lambda (x y) (list x y))
                              '(verb stem suffix) (list word stem suffix)))
          (push-last (list irreg 'error) *report*)))
        (progn
         (print-warning "Incorrectly formed verb. Presented ~s and produced ~{-s-^ ~}."
                        (first *word*)
                        (mapcan (lambda (x y) (list x y))
                              '(verb stem suffix) (list word stem suffix)))
         (push-last (list irreg 'error) *report*))))))

(defun past-tense (n &key (cont nil)(repfreq 100)(v nil))
  (unless cont
    (reset)
    (format t "~%" )
    (setf *report* nil)
    (setf *trial* 0 *repcount* 0))

  (sgp-fct (list :v v))

  (dotimes (i n)
    (add-past-tense-to-memory)
    (add-past-tense-to-memory)

```



Set a variable to the number of words from which to draw to get the right frequencies:

```
(defparameter *total-count* (caar (last *word-list*)))
```

Define a function to pick a random word from the set based on the relative frequencies:

```
(defun random-word ()
  (let ((num (act-r-random *total-count*)))
    (cdr (find-if (lambda (x) (< num (first x))) *word-list*))))
```

Make-one-goal picks a random verb and then creates a chunk that only has a value in the verb slot to place in the imaginal buffer and put the starting goal chunk in place:

```
(defun make-one-goal ()
  (setf *word* (random-word)))
```

Use the set-buffer-chunk command to place a chunk that is built with the define-chunks function (because we do not want to put it into declarative memory) into the **imaginal** buffer and then copy the starting-goal chunk into the **goal** buffer.

```
(set-buffer-chunk 'imaginal
  (car (define-chunks-fct
        (list (list 'verb (first *word*)))))
  (goal-focus starting-goal))
```

This function adds a correctly formed past tense to the declarative memory of the model by placing it into the **imaginal** buffer and then clearing the buffer so that the chunk gets merged into declarative memory normally.

```
(defun add-past-tense-to-memory ()
  (let ((word (random-word)))
    (set-buffer-chunk 'imaginal
      (car (define-chunks-fct
            (list (mapcan (lambda (x y) (list x y))
                          '(verb stem suffix) word))))
    (clear-buffer 'imaginal)))
```

The report-irreg function prints out the performance of the model averaged over every 1000 trials by default and draws the graph if the first optional parameter is true:

```
(defun report-irreg (&optional (graph nil) (trials 1000))
  (format t "~% Irreg  Reg  None  Overreg-%")
  (let ((data (mapcar 'fourth (rep-f-i *report* trials))))
    (when (and graph data)
      (graph-it data)))
  nil)
```

The graph-it function uses the experiment window generation tools to draw a graph of the model's performance for over generalized irregular verbs (the U-shaped learning):

```
(defun graph-it (data)
  (let* ((win (open-exp-window "Irregular Verbs correct" :width 500 :height 475))
        (low (apply 'min data)))
```

```

      (zoom (min .9 (/ (floor low .1) 10))))
(allow-event-manager win)
(clear-exp-window)
(add-text-to-exp-window :x 5 :y 5 :text "1.0" :width 22)
(add-text-to-exp-window :x 5 :y 400 :text (format nil "~0,2f" zoom) :width 22)
(add-text-to-exp-window :x 5 :y 200 :text (format nil "~0,2f"
                                          (+ zoom (/ (- 1.0 zoom) 2)))
                        :width 22)

(add-text-to-exp-window :x 200 :y 420 :text "Trials" :width 100)
(add-line-to-exp-window '(30 10) '(30 410) :color 'black)
(add-line-to-exp-window '(450 410) '(25 410) :color 'black)
(dotimes (i 10)
  (add-line-to-exp-window (list 25 (+ (* i 40) 10))
                          (list 35 (+ (* i 40) 10))
                          :color 'black))

(when (= (length data) 1)
  (push (first data) data))
(do* ((increment (max 1.0 (floor (/ 450.0 (length data))))))
      (range (floor 400 (- 1.0 zoom)))
      (intercept (+ range 10))
      (p1 (butlast data) (cdr p1))
      (p2 (cdr data) (cdr p2))
      (last-x 30 this-x)
      (last-y (- intercept (floor (* range (car p1))))
              (- intercept (floor (* range (car p1))))))
      (this-x (+ last-x increment)
              (+ last-x increment))
      (this-y (- intercept (floor (* range (car p2))))
              (- intercept (floor (* range (car p2))))))
      ((null (cdr p1)) (add-line-to-exp-window
                       (list last-x last-y) (list this-x this-y) :color 'red))
      (add-line-to-exp-window (list last-x last-y)
                              (list this-x this-y)
                              :color 'red))
(allow-event-manager win)))

```

The rep-f-i function does the analysis of the responses for output in report-irreg:

```

(defun rep-f-i (l n)
  (when l
    (let ((x (if (> (length l) n) (subseq l 0 n) l))
          (y (if (> (length l) n) (subseq l n) nil))
          (irreg 0)
          (reg 0)
          (none 0)
          (error 0))
      (dolist (i x)
        (when (first i)
          (case (second i)
            (reg
             (incf reg))
            (irreg
             (incf irreg))
            (none
             (incf none))
            (t
             (incf error))))))
    (let* ((total (+ irreg reg none))
           (data

```

```

      (if (zerop total)
          (list 0 0 0 0)
          (list (/ irreg total) (/ reg total)
                (/ none total) (if (> (+ irreg reg) 0) (/ irreg (+ irreg reg) 0))))))
    (format t "~{~6,3F~^ ~}~%" data)
    (cons data (rep-f-i y n))))))

```

The `add-to-report` function takes the name of a chunk and classifies the past tense that it contains, saves that classification for later analysis, and prints a warning if it's a badly formed past tense:

```

(defun add-to-report (the-chunk)
  (let ((stem (chunk-slot-value-fct the-chunk 'stem))
        (word (chunk-slot-value-fct the-chunk 'verb))
        (suffix (chunk-slot-value-fct the-chunk 'suffix))
        (irreg (eq (third *word*) 'blank)))

    (if (eq (first *word*) word)
        (cond
         ((and (eq stem word) (eq suffix 'ed))
          (push-last (list irreg 'reg) *report*))
         ((and (null suffix) (null stem))
          (push-last (list irreg 'none) *report*))
         ((and (eq stem (second *word*)) (eq suffix 'blank))
          (push-last (list irreg 'irreg) *report*))
         (t
          (print-warning "Incorrectly formed verb. Presented ~s and produced ~{~s~^ ~}."
                        (first *word*)
                        (mapcan (lambda (x y) (list x y))
                              '(verb stem suffix) (list word stem suffix)))
          (push-last (list irreg 'error) *report*)))
        (progn
         (print-warning "Incorrectly formed verb. Presented ~s and produced ~{~s~^ ~}."
                        (first *word*)
                        (mapcan (lambda (x y) (list x y))
                              '(verb stem suffix) (list word stem suffix)))
         (push-last (list irreg 'error) *report*))))))

```

The `past-tense` function presents a number of trials to the model. For every one that the model has to generate it first receives two correct ones into declarative memory.

```
(defun past-tense (n &key (cont nil)(repfreq 100)(v nil))
```

If the `cont` parameter is `nil` then the model needs to be reset and all of the data collected previously cleared:

```

  (unless cont
    (reset)
    (format t "~%" )
    (setf *report* nil)
    (setf *trial* 0 *repcount* 0))

```

Set the model's `:v` parameter based on the parameter passed to `past-tense` using the functional form of `sgp`:

```
(sgp-fct (list :v v))
```

For n trials present the model with two correct past tenses and then run it to generate one:

```
(dotimes (i n)
```

Add two past tenses to declarative memory

```
(add-past-tense-to-memory)  
(add-past-tense-to-memory)
```

Generate a new goal and verb for the imaginal buffer and run the model up to 200 seconds

```
(make-one-goal)  
(run 200)
```

Record the model's response and print the data every repfreq trials

```
(add-to-report (buffer-read 'imaginal))  
(clear-buffer 'imaginal)  
(incf *repcount*)  
(incf *trial*)  
(when (>= *repcount* repfreq)  
  (format t "Trial ~6D : " *trial*)  
  (rep-f-i (subseq *report* (- *trial* repfreq)) repfreq)  
  (setf *repcount* 0))
```

Run the model for 200 seconds before presenting the next trial

```
(run-full-time 200))
```

The first two new commands used in this unit are functions for manipulating the contents of buffers.

**set-buffer-chunk** is a function that can be used to set a buffer to hold a copy of a particular chunk. It takes two parameters which must be the name of a buffer and the name of a chunk respectively. It copies that chunk into the specified buffer (clearing any chunk which may have been in the buffer at that time) and returns the name of the chunk which is now in the buffer. This acts very much like the goal-focus command, except that it works for any buffer. An important difference however is that the goal-focus command actually schedules an event which uses set-buffer-chunk to put the chunk in the goal buffer. That is important because scheduled events display in the model trace and are changes to which the model can react. Typically, the schedule-set-buffer-chunk command is more appropriate than set-buffer-chunk for that reason and you can find the details on that command in the reference manual.

**clear-buffer** is a function that can be used to clear a chunk from a buffer the same way a *-buffer>* action in a production will. It takes one parameter which should be the name of a buffer and that buffer is emptied and the chunk merged into declarative memory.

The other two new commands in this unit are more general purpose tools.

**push-last** is a macro for adding an item to the end of a list. It takes two parameters, any item and a list. It destructively adds the item to the end of the list.

**print-warning** is a macro for displaying warnings messages. It takes parameters similar to the format command (a format string and then any number of parameters after that). It will use that format string to display the parameters provided inside of a block quote which starts with “Warning:”. It will print the output regardless of the settings of the :v and :cmdt parameters.