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 *number*)
(defvar *word*)
(defvar *repcount*)
(defvar *trial*)

(defun make-triples (l)
   (when l
   (cons (list (first l)(second l)(third l)
               (fourth l)
               (if (eq (second l) 'I) 'blank 'ed))
         (make-triples (nthcdr 4 l)))))

(defparameter *word-list*
  (make-triples '(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* (apply #'+ (mapcar #'third *word-list*)))

;;; Select a random word from the vocabulary, but based on its frequency

(defun random-word ()
  (let
    ((num (act-r-random *total-count*)))
    (dolist (i *word-list*)
      (if (< num (third i))
        (return i)
        (setf num (- num (third i)))))))

;;; Set the goal to do one past tense

(defun make-one-goal ()
   (let*
```

```
     ((wordpair (random-word))
      (word (first wordpair))
      (word-no (second wordpair)))

     (setf *number* word-no *word* word)
     (set-buffer-chunk 'imaginal (car (define-chunks-fct
                                        (list (list 'isa 'past-tense 'verb word)))))
     (goal-focus starting-goal)))

;;; This function simulates "hearing" a past tense: it adds a correct
;;; past tense to memory

(defun add-past-tense-to-memory ()
   (let*
     ((wordpair (random-word))
      (word (first wordpair))
      (stem (fourth wordpair))
      (suffix (fifth wordpair)))
     (set-buffer-chunk 'imaginal (car (define-chunks-fct
                                       (list (list 'isa 'past-tense 'verb word
                                                   'stem stem 'suffix suffix)))))
     (clear-buffer 'imaginal)))


;;; The following function reports how often an irregular word gets an irregular
;;; (correct), regular past tense or just the stem as past tense (None). It
;;; shows how this developes in time.

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

(defun graph-it (data)
  (let ((win (open-exp-window "Irregular Verbs correct" :width 400 :height 350)))
    (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 300 :text "0.7" :width 22)
    (add-text-to-exp-window :x 150 :y 320 :text "Trials" :width 100)
    (add-line-to-exp-window '(30 10) '(30 310) :color 'black)
    (add-line-to-exp-window '(380 310) '(30 310) :color 'black)
    (add-line-to-exp-window '(25 10) '(35 10) :color 'black)
    (add-line-to-exp-window '(25 110) '(35 110) :color 'black)
    (add-line-to-exp-window '(25 210) '(35 210) :color 'black)

    (do* ((increment (max 1.0 (floor (/ 350.0 (length data)))))
          (p1 (butlast data) (cdr p1))
          (p2 (cdr data) (cdr p2))
          (last-x 30 this-x)
          (last-y (+ 10 (floor (* (- 1.0 (car p1)) 1000)))
                  (+ 10 (floor (* (- 1.0 (car p1)) 1000))))
          (this-x (+ last-x increment)
                  (+ last-x increment))
          (this-y (+ 10 (floor (* (- 1.0 (car p2)) 1000)))
                  (+ 10 (floor (* (- 1.0 (car p2)) 1000)))))
         ((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))))
```

```
(defun rep-f-i (l n)
   (if 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)
           (data nil))
       (dolist (i x)
         (cond ((eq (first i) 'R) nil)
               ((eq (second i) 'reg) (setf reg (1+ reg)))
               ((eq (second i) 'irreg) (setf irreg (1+ irreg)))
               (t (setf none (1+ none)))))
       (if (> (+ irreg reg none) 0)
         (setf data (list (/ irreg (+ irreg reg none))
                  (/ reg (+ irreg reg none))(/ none (+ irreg reg none))
                  (if (> (+ irreg reg) 0) (/ irreg (+ irreg reg)) 0)))
         (setf data (list 0 0 0 0)))
       (format t "~{~6,3F~}~%" data)
       (cons data (rep-f-i y n)))
    nil))

(defun add-to-report (goal-chunk)
   (let* ((stem (chunk-slot-value-fct goal-chunk 'stem))
          (word (chunk-slot-value-fct goal-chunk 'verb))
          (suffix (chunk-slot-value-fct goal-chunk 'suffix)))
     (cond
      ((eq stem word) (push (list *number* 'reg *word*) *report*))
      ((eq suffix nil) (push (list *number* 'none *word*) *report*))
      (t (push (list *number* 'irreg *word*) *report*)))))


;;; This function will run the experiment for n trials.
;;; The keyword parameters are:
;;;   cont - continue, when set to true the experiment continues instead of
;;;          starting anew.
;;;   repfreq - determines how often results are reported during a run.
;;;   v - the setting for the :v parameter of the model i.e. whether or not
;;;       to display the model's trace.

(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)
    (make-one-goal)
    (run 200)
    (add-to-report (buffer-read 'imaginal))
    (clear-buffer 'imaginal)
    (incf *repcount*)
    (when (>= *repcount* repfreq)
      (format t "Trial ~6D : " (1+ *trial*))
      (rep-f-i (subseq *report* 0 repfreq) repfreq)
      (setf *repcount* 0))
```

```
    (run-full-time 200)
    (incf *trial*)))
```

Here is the detailed description.

Start by defining some global variables:

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

Define a function to build the word list for use in the experiment. The list will encode the verb along with its type, frequency, and correct past tense.

```
 (defun make-triples (l)
   (when l
   (cons (list (first l)(second l)(third l)
               (fourth l)
               (if (eq (second l) 'I) 'blank 'ed))
         (make-triples (nthcdr 4 l)))))
```

Construct the list of words using the make-triples function:

```
(defparameter *word-list*
  (make-triples '(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)))
```

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

```
(defparameter *total-count* (apply #'+ (mapcar #'third *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*)))
    (dolist (i *word-list*)
      (if (< num (third i))
        (return i)
        (setf num (- num (third i)))))))
```

Create a new past-tense chunk with a verb that the model should build a past tense of and put the starting goal chunk in place:

```
(defun make-one-goal ()
   (let*
     ((wordpair (random-word))
      (word (first wordpair))
      (word-no (second wordpair)))

     (setf *number* word-no *word* 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 'isa 'past-tense 'verb 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*
     ((wordpair (random-word))
      (word (first wordpair))
      (stem (fourth wordpair))
      (suffix (fifth wordpair)))
     (set-buffer-chunk 'imaginal (car (define-chunks-fct
                                        (list (list 'isa 'past-tense 'verb word
                                                    'stem stem 'suffix suffix)))))
     (clear-buffer 'imaginal)))
```

The report-irreg function prints out the performance of the model averaged over every 1000 trials by default:

```
(defun report-irreg (&optional (graph nil) (trials 1000))
   (format t "~% Irreg    Reg   None   Overreg~%")
   (let ((data (mapcar #'fourth (rep-f-i (reverse *report*) trials))))
     (when graph
       (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 400 :height 350)))
    (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 300 :text "0.7" :width 22)
    (add-text-to-exp-window :x 150 :y 320 :text "Trials" :width 100)
    (add-line-to-exp-window '(30 10) '(30 310) :color 'black)
    (add-line-to-exp-window '(380 310) '(30 310) :color 'black)
    (add-line-to-exp-window '(25 10) '(35 10) :color 'black)
    (add-line-to-exp-window '(25 110) '(35 110) :color 'black)
    (add-line-to-exp-window '(25 210) '(35 210) :color 'black)

    (do* ((increment (max 1.0 (floor (/ 350.0 (length data)))))
          (p1 (butlast data) (cdr p1))
          (p2 (cdr data) (cdr p2))
          (last-x 30 this-x)
          (last-y (+ 10 (floor (* (- 1.0 (car p1)) 1000)))
                  (+ 10 (floor (* (- 1.0 (car p1)) 1000))))
          (this-x (+ last-x increment)
                  (+ last-x increment))
          (this-y (+ 10 (floor (* (- 1.0 (car p2)) 1000)))
                  (+ 10 (floor (* (- 1.0 (car p2)) 1000)))))
        ((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))))
```

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

```
(defun rep-f-i (l n)
   (if 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)
           (data nil))
       (dolist (i x)
         (cond ((eq (first i) 'R) nil)
               ((eq (second i) 'reg) (setf reg (1+ reg)))
               ((eq (second i) 'irreg) (setf irreg (1+ irreg)))
               (t (setf none (1+ none)))))
       (if (> (+ irreg reg none) 0)
         (setf data (list (/ irreg (+ irreg reg none))
                  (/ reg (+ irreg reg none))(/ none (+ irreg reg none))
                  (if (> (+ irreg reg) 0) (/ irreg (+ irreg reg)) 0)))
         (setf data (list 0 0 0 0)))
       (format t "~{~6,3F~}~%" data)
       (cons data (rep-f-i y n)))
    nil))
```

The add-to-report function takes the name of a chunk and records the result of the past tense that it contains for later analysis:

```
(defun add-to-report (goal-chunk)
   (let* ((stem (chunk-slot-value-fct goal-chunk 'stem))
          (word (chunk-slot-value-fct goal-chunk 'verb))
          (suffix (chunk-slot-value-fct goal-chunk 'suffix)))
     (cond
```

```
    ((eq stem word) (push (list *number* 'reg *word*) *report*))
    ((eq suffix nil) (push (list *number* 'none *word*) *report*))
    (t (push (list *number* 'irreg *word*) *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 and clear all of the data collection variables:

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

Set the model's :v parameter based on the parameter passed to do-it 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 to generate a new past tense and run the model

```
    (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*)
    (when (>= *repcount* repfreq)
      (format t "Trial ~6D : " (1+ *trial*))
      (rep-f-i (subseq *report* 0 repfreq) repfreq)
      (setf *repcount* 0))
```

Run the model for 200 seconds before presenting the next trial

```
    (run-full-time 200)
    (incf *trial*)))
```

The 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.