# Unit 7 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.

## Lisp

Start by loading the starting model for this task.

```
(load-act-r-model "ACT-R:tutorial;unit7;past-tense-model.lisp")
```

Define some global variables to hold the collected data and generate the verbs with the appropriate frequencies.

```
(defvar *report*)
(defvar *total-count*)
(defvar *word-list*)

(defparameter *verbs* '((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)))
```

Define a function to build the word list for use in the experiment. The list for each verb will include a count for generating a random verb based on its frequency in English, the verb, and its correct past tense.

```
(defun make-word-freq-list (l)
  (let ((data nil)
        (count 0))
    (dolist (verb l)
      (incf count (third verb))
      (push-last (list count (first verb) (fourth verb)
                       (if (eq (second verb) 'i) 'blank 'ed))
            data))
    (setf *total-count* count)
    data))
```

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 from the list.

```
(defun make-one-goal ()
  (let ((word (random-word)))
```

Then it creates a chunk that only has the base form of that random verb in the verb slot and places it in the **imaginal** buffer using the set-buffer-chunk command.

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

It places a copy of the starting-goal chunk into the **goal** buffer and returns the word being presented.

```
    (goal-focus starting-goal)
    word))
```

The add-past-tense-to-memory 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 print-header function just prints the column labels for the data output.

```
(defun print-header ()
  (format t "~%trials      Irregular       Regular    No inflection  Inflected correctly~%"))
```

The past-tense-results function prints out the performance of the model averaged over every 1000 trials and optionally draws a graph of the results if the optional parameter is true (which it is by default).

```
(defun past-tense-results (&optional (graph t))
  (print-header)
  (let ((data (rep-f-i 0 (length *report*) 1000)))
    (when (and graph (> (length data) 1))
      (graph-it data))
    data))
```

The graph-it function opens an experiment window and draws a graph of the model's correct performance for inflected irregular verbs (where the U-shaped learning should appear).

```
(defun graph-it (data)
  (let* ((win (open-exp-window "Irregular Verbs correct" :visible t :width 500 :height 475))
         (low (apply 'min data))
         (zoom (min .9 (/ (floor low .1) 10))))

    (clear-exp-window win)
    (add-text-to-exp-window win :text "1.0" :x 5 :y 5 :width 22)
    (add-text-to-exp-window win :text (format nil "~0,2f" zoom) :x 5 :y 400 :width 22)
    (add-text-to-exp-window win :text (format nil "~0,2f" (+ zoom (/ (- 1.0 zoom) 2)))
                            :x 5 :y 200 :width 22)

    (add-text-to-exp-window win "Trials" 200 420 100)
    (add-line-to-exp-window win '(30 10) '(30 410) 'black)
    (add-line-to-exp-window win '(450 410) '(25 410) 'black)

    (dotimes (i 10)
      (add-line-to-exp-window win (list 25 (+ (* i 40) 10)) (list 35 (+ (* i 40) 10)) 'black))

    (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 win
                            (list last-x last-y) (list this-x this-y) 'red))
      (add-line-to-exp-window win (list last-x last-y)
                               (list this-x this-y)
                               'red))))
```

The safe-div function returns the result of n/d unless d is zero in which case it returns 0.

```
(defun safe-div (n d)
  (if (zerop d)
      0
    (/ n d)))
```

The rep-f-i function computes the average performance from a specified portion of the model's data in segments of the given length and prints out the lines of data and returns the list of the inflected irregular verb results.

```
(defun rep-f-i (start end count)
  (let ((data nil))
    (dotimes (i (ceiling (- end start) count))
      (let ((irreg 0)
            (reg 0)
            (none 0)
            (e (if (> end (+ start count (* i count)))
                   (+ start count (* i count))
                 end)))

        (dolist (x (subseq *report* (+ start (* i count)) e))
          (when (first x)
            (case (second x)
              (reg
               (incf reg))
              (irreg
               (incf irreg))
              (none
               (incf none)))))

        (let* ((total (+ irreg reg none))
               (correct (safe-div irreg (+ irreg reg))))
          (format t "~6d ~{~13,3F~^ ~}~%" e
            (list
             (safe-div irreg total)
             (safe-div reg total)
             (safe-div none total)
             correct))
          (push-last correct data))))
    data))
```

The add-to-report function takes a verb list and the name of a chunk which the model generated. It classifies the generated chunk based on how it formed the past tense and records that result in the global report. It also prints out warnings if the verb is malformed in anyway.

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

    (if (eq (first target) 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 target)) (eq suffix 'blank))
               (push-last (list irreg 'irreg) *report*))
              (t
               (print-warning
                 (format nil "Incorrectly formed verb.  Presented ~s and produced ~{~s~^ ~}."
                             (first target)
                             (mapcan (lambda (x y) (list x y))
                               '(verb stem suffix) (list word stem suffix))))
               (push-last (list irreg 'error) *report*)))
      (progn
        (print-warning
          (format nil "Incorrectly formed verb.  Presented ~s and produced ~{~s~^ ~}."
                      (first target)
                      (mapcan (lambda (x y) (list x y))
                        '(verb stem suffix) (list word stem suffix))))
        (push-last (list irreg 'error) *report*)))))
```

Define another global variable which is used to check whether the model receives a reward, and then a function which will set that when a reward is given.

```
(defvar *reward-check*)

(defun verify-reward (&rest r)
  (declare (ignore r))
  (setf *reward-check* t))
```

The past-tense-trials function takes one required parameter and two optional ones. The required parameter specifies how many verbs the model will be given to create a past-tense. If the first optional parameter is provided as true then the model should continue from where it left off otherwise it will be reset before presenting the verbs. If the third optional parameter is given as true then the model trace will be shown while it runs, otherwise it will be turned off. It will print the averaged results every 100 trials as it runs.

```
(defun past-tense-trials (n &optional (cont nil)(v nil))
```

Add the command that will set the flag to indicate a reward was received and have it monitor the trigger-reward command.

```
(add-act-r-command "reward-check" 'verify-reward
                   "Past tense code check for a reward each trial.")
(monitor-act-r-command "trigger-reward" "reward-check")
```

If the cont parameter is nil or the word list has not been created then the model needs to be reset and all of the global data initialized, making sure to create chunks that represent all of the verbs if they are not already chunks.

```
(when (or (null *word-list*) (null cont))
  (reset)
  (setf *word-list* (make-word-freq-list *verbs*))
  (let ((new nil))
    (dolist (x *word-list*)
      (mapcar (lambda (y) (pushnew y new)) (cdr x)))

    (dolist (x new)
      (unless (chunk-p-fct x)
        (define-chunks-fct (list (list x))))))

  (print-header)
  (setf *report* nil))
```

Set the value of the model’s :v parameter based on the value provided to turn the trace off or on.

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

Loop over the n trials, adding two correct past tenses to the models memory before presenting a verb for it to generate. Run it for up to 100 seconds to generate the past tense, add that result to the data, clear the imaginal buffer so the current result can enter declarative memory, check whether it should print the results, and then print a warning if the model did not receive a reward or if it actually spent the entire 100 seconds trying to make the past tense.

```
(let* ((start (* 100 (floor (length *report*) 100)))
       (count (mod (length *report*) 100)))
  (dotimes (i n)
    (add-past-tense-to-memory)
    (add-past-tense-to-memory)
    (setf *reward-check* nil)
    (let ((target (make-one-goal))
          (duration (run 100)))
      (add-to-report target (buffer-read 'imaginal))
      (clear-buffer 'imaginal)
      (incf count)
      (when (= count 100)
        (rep-f-i start (+ start 100) 100)
        (setf count 0)
        (incf start 100))
      (unless *reward-check*
        (print-warning
         "Model did not receive a reward when given ~s."
         (first target)))
      (run-full-time (- 200 duration))

      (when (= duration 100)
        (print-warning
         "Model spent 100 seconds generating a past tense for ~s."
         (first target)))))
  (rep-f-i start (+ start count) 100))
```

Remove the monitor for trigger-reward when done.

```
(remove-act-r-command-monitor "trigger-reward" "reward-check")
(remove-act-r-command "reward-check")
nil)
```

## Python

Start by importing the modules needed and loading the starting model for this task.

```
import actr
import math

actr.load_act_r_model("ACT-R:tutorial;unit7;past-tense-model.lisp")
```

Define some global variables to hold the collected data and generate the verbs with the appropriate frequencies.

```
report = []
total_count = 0
word_list = []

verbs = [['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']]
```

Define a function to build the word list for use in the experiment. The list for each verb will include a count for generating a random verb based on its frequency in English, the verb, and its correct past tense.

```
def make_word_freq_list (l):

    data = []
    count = 0

    for verb in l:
        count += verb[2]
        if verb[1] == 'i':
            suffix = 'blank'
        else:
            suffix = 'ed'

        data.append([count,verb[0],verb[3],suffix])
    global total_count
    total_count = count
    return(data)
```

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

```
def random_word():

    num=actr.random(total_count)

    for i in word_list:
        if i[0] > num:
            return(i[1:])
```

Make_one_goal picks a random verb from the list.

```
def make_one_goal():

    word = random_word()
```

Then it creates a chunk that only has the base form of that random verb in the verb slot and places it in the **imaginal** buffer using the set_buffer_chunk command.

```
    actr.set_buffer_chunk('imaginal',actr.define_chunks(['verb',word[0]])[0])
```

It places a copy of the starting-goal chunk into the **goal** buffer and returns the word being presented.

```
    actr.goal_focus('starting-goal')

    return(word)
```

The add_past_tense_to_memory 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.

```
def add_past_tense_to_memory ():

    word = random_word()

    actr.set_buffer_chunk('imaginal',
                          actr.define_chunks(['verb',word[0],
                                              'stem',word[1],
                                              'suffix',word[2]])[0])
    actr.clear_buffer('imaginal')
```

The print_header function just prints the column labels for the data output.

```
def print_header():
    print ()
    print ( "trials      Irregular       Regular    No inflection  Inflected correctly")
```

The results function prints out the performance of the model averaged over every 1000 trials and optionally draws a graph of the results if the optional parameter is true (which it is by default).

```
def results(graph=True):

    print_header()
    data = rep_f_i(0,len(report),1000)
    if graph and len(data) > 1:
        graph_it(data)
    return(data)
```

The graph_it function opens an experiment window and draws a graph of the model's correct performance for inflected irregular verbs (where the U-shaped learning should appear).

```
def graph_it(data):

    win = actr.open_exp_window("Irregular Verbs correct", visible=True,height=500,width=475)
    low = min(data)
    zoom = min([.9,math.floor(10 * low)/10])

    actr.clear_exp_window(win)
    actr.add_text_to_exp_window(win,text="1.0",x=5,y=5,width=22)
    actr.add_text_to_exp_window(win,text="%0.2f" % zoom,x=5,y=400,width=22)
    actr.add_text_to_exp_window(win,text="%0.2f" % (zoom + ((1.0 - zoom) / 2)),
                                x=5,y=200,width=22)

    actr.add_text_to_exp_window(win,"Trials",200,420,100)
    actr.add_line_to_exp_window(win,[30,10],[30,410],'black')
    actr.add_line_to_exp_window(win,[450,410],[25,410],'black')

    for i in range(10):
        actr.add_line_to_exp_window(win,[25,10 + i*40],[35,10 + i*40],'black')

    start = data[0]
    increment = max([1.0,math.floor( 450 / len(data))])
    r = math.floor (400/ (1.0 - zoom))
    intercept = r + 10
    lastx =30
    lasty = intercept - math.floor(r * start)

    for p in data[1:]:
        x = lastx + 30
        y = intercept - math.floor(r * p)

        actr.add_line_to_exp_window(win,[lastx,lasty],[x,y],'red')
        lastx = x
        lasty = y
```

The safe_div function returns the result of n/d unless d is zero in which case it returns 0.

```
def safe_div(n, d):
    if d == 0:
        return(0)
    else:
        return (n / d)
```

The rep_f_i function computes the average performance from a specified portion of the model's data in segments of the given length and prints out the lines of data and returns the list of the inflected irregular verb results.

```
def rep_f_i(start,end,count):

    data = []

    for i in range(math.ceil( (end - start) / count)):
        irreg = 0
        reg = 0
        none = 0

        if end > (start + count + (i * count)):
            e = start + count + (i * count)
        else:
            e = end

        for x in report[(start + (i * count)):e]:
            if x[0]:
                if x[1] == 'reg':
                    reg += 1
                elif x[1] == 'irreg':
                    irreg += 1
                elif x[1] == 'none':
                    none += 1

        total = irreg + reg + none
        correct = safe_div(irreg, (irreg + reg))

        print("%6d %13.3f %13.3f %13.3f %13.3f"%
              (e,safe_div(irreg,total),safe_div(reg,total),safe_div(none,total),correct))

        data.append(correct)

    return data
```

The add_to_report function takes a verb list and the name of a chunk which the model generated. It classifies the generated chunk based on how it formed the past tense and records that result in the global report. It also prints out warnings if the verb is malformed in anyway.

```
def add_to_report(target, chunk):
    global report

    stem = actr.chunk_slot_value(chunk,"stem")
    word = actr.chunk_slot_value(chunk,"verb")
    suffix = actr.chunk_slot_value(chunk,"suffix")
    irreg = (target[2] == 'blank')
    if target[0].lower() == word.lower():
        if stem == word and suffix.lower() == 'ed':
            report.append([irreg,'reg'])
        elif stem == None and suffix == None:
            report.append([irreg,'none'])
        elif stem.lower() == target[1].lower() and suffix.lower() == 'blank':
            report.append([irreg,'irreg'])
        else:
            actr.print_warning(
              "Incorrectly formed verb. Presented %s and produced verb %s,stem %s,suffix %s."%
              (target[0],word,stem,suffix))
    else:
        actr.print_warning(
            "Incorrectly formed verb. Presented %s and produced verb %s,stem %s,suffix %s."%
            (target[0],word,stem,suffix))
        report.append([irreg,'error'])
```

Define another global variable which is used to check whether the model receives a reward, and then a function which will set that when a reward is given.

```
reward_check = False

def verify_reward(*params):
    global reward_check
    reward_check = True
```

The trials function takes one required parameter and two optional ones. The required parameter specifies how many verbs the model will be given to create a past-tense. If the first optional parameter is provided as true then the model should continue from where it left off otherwise it will be reset before presenting the verbs. If the third optional parameter is given as true then the model trace will be shown while it runs, otherwise it will be turned off. It will print the averaged results every 100 trials as it runs.

```
def trials(n,cont=False,v=False):

    global report,word_list,reward_check
```

Add the command that will set the flag to indicate a reward was received and have it monitor the trigger-reward command.

```
    actr.add_command("reward-check",verify_reward,
                     "Past tense code check for a reward each trial.")
    actr.monitor_command("trigger-reward","reward-check")
```

If the cont parameter is False or the word list has not been created then the model needs to be reset and all of the global data initialized, making sure to create chunks that represent all of the verbs if they are not already chunks.

```
    if not(cont) or not(word_list):
        actr.reset()
        word_list = make_word_freq_list(verbs)
        new = []
        for x in word_list:
            for y in x[1:]:
```

```
            if y not in new:
                new.append(y)
    for x in new:
        if not(actr.chunk_p(x)):
            actr.define_chunks([x])

    print_header()
    report = []
```

Set the value of the model's :v parameter based on the value provided to turn the trace off or on.

```
actr.set_parameter_value(":v",v)
```

Loop over the n trials, adding two correct past tenses to the models memory before presenting a verb for it to generate. Run it for up to 100 seconds to generate the past tense, add that result to the data, clear the imaginal buffer so the current result can enter declarative memory, check whether it should print the results, and then print a warning if the model did not receive a reward or if it actually spent the entire 100 seconds trying to make the past tense.

```
start = 100 * math.floor(len(report) / 100)
count = len(report) % 100

for i in range(n):
    add_past_tense_to_memory()
    add_past_tense_to_memory()
    reward_check = False
    target = make_one_goal()
    duration = actr.run(100)[0]
    add_to_report(target,actr.buffer_read('imaginal'))
    actr.clear_buffer('imaginal')
    count += 1

    if count == 100:
        rep_f_i(start, start + 100, 100)
        count = 0
        start += 100
    if not(reward_check):
        actr.print_warning("Model did not receive a reward when given %s."% target[0])

    actr.run_full_time(200 - duration)

    if duration == 100:
        actr.print_warning("Model spent 100 seconds generating a past tense for %s."%
                           target[0])

rep_f_i(start,start+count,100)
```

Remove the monitor for trigger-reward when done.

```
actr.remove_command_monitor("trigger-reward","reward-check")
actr.remove_command("reward-check")
```

## New Commands

**set-buffer-chunk** and **set_buffer_chunk** can be used to set a buffer to hold a copy of a particular chunk. They take 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. There is another command which works more like goal-focus which is often more appropriate (schedule-set-buffer-chunk the details of which you can find in the reference manual), but this experiment's code doesn't do that in either place where the command is used. In the first instance it's because it is putting a chunk into the buffer and then immediately removing it without needing to run the model, and in the other instance putting a chunk into the buffer is also accompanied by a goal-focus call which generates an event to which the model can respond so it's not necessary to create another event (although it would not affect anything in this task if it did do it the "right" way by scheduling the change).

**clear-buffer** and **clear_buffer** 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 is merged into declarative memory. It returns the name of the chunk that was cleared. Like set-buffer-chunk, the clear-buffer command does not create an event to which the model can respond, and there is a corresponding schedule-clear-buffer which could be used to do that.

**chunk-p** and **chunk_p** can be used to determine if the provided name is actually the name of a chunk in the current model. They return a true result if it is (t/True) and false if not (nil/None).

**print-warning** and **print_warning** can be used for outputting information in the ACT-R warning trace. The Python function takes a single parameter which is a string and prints that as an ACT-R warning message. The Lisp version is a little more general and works similar to the Lisp format command. It takes a format string and then any number of parameters after that to be used in the format string.

**push-last** is a macro for Lisp which adds 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. This is often more convenient than the standard Lisp function push which adds to the beginning of a list.

**trigger-reward** and **trigger_reward** were not actually used in the code, but it was monitored to determine if the model received a reward. This command can be called to provide a reward for utility learning to the model at any time, and it schedules an event to provide that reward at the current time. It requires one parameter which indicates the reward. If that reward value is numeric then it is used to update the utilities of those productions which have been selected since the last reward. If it is not a numeric value then it only serves to mark the last reward time for preventing further propagation of the next reward signal.