# Unit 7: Production Rule Learning

In this unit we will discuss how new production rules are learned. As we will see, a model can acquire new production rules by collapsing two production rules that apply in succession into a single rule. Through this process the model will transform knowledge that is stored declaratively into a procedural form. We call this process of forming new production rules **production compilation** and refer to the specific act of combining two productions as a composition.

## 7.1 The Basic Idea

A good pair of productions for illustrating production compilation is the two that fire in succession to retrieve a paired associate in the **paired** model from Unit 4:

```
(p read-probe
    =goal>
      isa      goal
      state    attending-probe
    =visual>
      isa      visual-object
      value    =val
    ?imaginal>
      state    free
   ==>
    +imaginal>
      isa      pair
      probe    =val
    +retrieval>
      isa      pair
      probe    =val
    =goal>
      state    testing
)
```

```
(p recall
    =goal>
      isa      goal
      state    testing
    =retrieval>
      isa      pair
      answer   =ans
    ?manual>
      state    free
    ?visual>
      state    free
   ==>
    +manual>
      cmd      press-key
      key      =ans
    =goal>
      state    read-study-item
    +visual>
      cmd      clear
)
```

If these two productions fired and retrieved the chunk for the pair of zinc & 9, production compilation would compose these two rules into the following single production:

```
(P PRODUCTION0
  "READ-PROBE & RECALL - CHUNK0-0"
   =GOAL>
       STATE ATTENDING-PROBE
   =VISUAL>
       VALUE "zinc"
   ?IMAGINAL>
       STATE FREE
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
```

```
 ==>
   =GOAL>
       STATE READ-STUDY-ITEM
   +VISUAL>
       CMD CLEAR
   +MANUAL>
       CMD PRESS-KEY
       KEY "9"
   +IMAGINAL>
       PROBE "zinc"
)
```

Essentially, this production combines the work of the two and has built into it the paired associate. In the next two subsections we will describe the general principles used for composing two production rules together and the factors that control how these productions compete in the conflict resolution process.

## 7.2 Forming a New Production

The basic idea behind forming a new production is to combine the tests in the two conditions into a single set of tests that will recognize when the pair of productions will apply and combine the two sets of actions into a single set of actions that has the same overall effect. Since the conditions consist of a set of buffer tests and the actions consist of a set of buffer transformations (either direct modifications or new requests) this can be done largely on a buffer-by-buffer basis. The complications occur when there is a buffer transformation in the action of the first production and either a test of that buffer in the condition of the second production or another transformation of the same buffer in the action of the second production. The productions above illustrate both complications with respect to the **goal** buffer. Because the change to the state slot of the **goal** buffer chunk in the first production is tested as a condition in the second production that test can be omitted from the tests of the composed production. Then, because that state slot is changed again by the second production, and the composed production only needs to reproduce the final state of the original two productions, that first goal change can also be omitted from the actions of the compiled production. The result of the overlap in the **goal** buffer is just a simplification of the production rule but in other cases other responses are necessary.

Because different modules use their buffers in different ways the production compilation process needs to be sensitive to those differences. For instance, in the above production we see that the **retrieval** buffer request was omitted from the newly formed production, but the **imaginal** buffer request was not. The production compilation mechanism is built around a set of buffer "compilation types" and each buffer is classified as one of the possible compilation types. For each compilation type there is a set of rules that specify when two productions that use such a buffer can be combined through compilation, and for each type there is a set of rules for how to combine the uses of a buffer. By default there are five types to which the buffers of ACT-R are assigned and we will describe the mechanisms used for those types in the following sections. It is possible to add new

types and to adjust the assignment of buffers to types, but that is beyond the scope of the tutorial.

### 7.2.1 Motor Type Buffers

Let us first consider the compilation policy for the motor type buffers. The buffers that fit this type are the **manual** and **vocal** buffers. The main distinction of these buffers is that they never hold a chunk. They are used for requesting actions to a module which can only process one request at a time and they are only tested through queries. If the first production makes a request of one of these buffers then it is not possible to compose it with a second production if that production also makes a request of that buffer or queries the buffer for anything other than state busy. If both productions make a request, then there is a danger of jamming if both requests were to occur at the same time, and any queries that are not checking to see if the module is busy in the second production are probably there to prevent jamming in the future, so also block the composition.

### 7.2.2 Perceptual Type Buffers

Now let us consider the compilation policy for the perceptual buffers. These are the buffers in the perceptual type: **visual-location**, **visual**, **aural-location**, and **aural.** These buffers will hold chunks generated by their modules. The important characteristic about them is that those chunks are based on information in the external world, and thus are not guaranteed to result in the same request generating the same result at another time. First, like the motor type buffers, it is not possible to compose two productions which both make requests of the same perceptual type buffer or if the first production makes a request and the second production makes a query for other than state busy of that buffer because of the possibility of jamming. In addition, if the first production makes a request of one of these buffers then it is not possible to compose it with the second production if that production tests the contents of the buffer. This is because of the unpredictable nature of such requests – one does not want to create productions that encapsulate information that is based on external information which may not be valid ever again (at least not for most modeling purposes). The idea is that we only want to create new productions that are "safe", and by safe we mean that the new production can only match if the productions that it was generated from would match and that its actions are the same as those of its parent productions. Basically, for the default mechanism, we do not want composed productions to be generated that introduce new errors into the model.

Thus, points where a request is made of a perceptual or motor type buffer are points where there are natural breaks in the compilation process. The standard production compilation mechanisms will not compose a production that makes such a request with a following production that operates on the same buffer.

### 7.2.3 Retrieval Type Buffer

Next let us consider the compilation policy for the **retrieval** buffer (the only buffer of the retrieval type). Because declarative memory is an internal mechanism (i.e., not subject to the whims of the outside world) it is more predictable and thus offers an opportunity for economy. The interesting opportunity for economy occurs when the first production requests a retrieval and the second tests the result of that retrieval. In this case, one can delete the request and test and instead specialize the composed production. Any variables specified in the retrieval request and bound in the harvesting of the chunk can be replaced with the specific values based on the chunk that was actually retrieved. This was what happened in the example production above where the retrieved paired-associate for zinc & 9 was built into the new production0. There is one case, however, which blocks the composition of a production which makes a retrieval request with a subsequent production. That is when the second production has a query for a retrieval failure. This cannot be composed because declarative memory grows monotonically and it is not safe to predict that in the future there will be a retrieval failure. This suggests that it is preferable, if possible, to write production rules that do not depend on retrieval failures.

### 7.2.4 Goal and Imaginal Type Buffers

The **goal** and **imaginal** buffers are also internal buffers allowing economies to be achieved. The mechanisms used for these two buffers are very similar, and thus will be described together. The difference between them arises from the fact that the **imaginal** buffer requests take time to complete, and that difference will be described below. First, we will analyze the process for which they are the same and that is broken down into cases based on whether the first production involves a request to the buffer or not.

#### *7.2.4.a First production does not make a request*

Let C1 and C2 be the conditions for the buffer in the first and second production and A1 and A2 be the corresponding productions' buffer modification actions for that buffer. Then, the buffer test for the composed production is C1+(C2-A1) where (C2-A1) specifies those things tested in C2 that were not created in A1. The modification for the combined production is A2+(A1~A2) where (A1~A2) indicates those things that were in A1 that are not undone by A2. If the second production makes a request, then that request can just be included in the composed production.

#### *7.2.4.b First production makes a request*

This case breaks down into two subcases depending on whether the second production also makes a request.

**The second production does not also make a request.** In this case the second production's buffer test can be deleted since its satisfaction is guaranteed by the first production. Let C1 be the buffer condition of the first production, A1 be the buffer modification action of the first production, N1 be the new request in the first production,

and A2 be the buffer modification of the second production. Then the buffer test of the composed production is just C1, the goal modification is just A1, and the new request is A2+(N1~A2).

**The second production also makes a request.** In this case the two productions cannot be composed because this would require either skipping over the intermediate request, which would result in a chunk not being created in the new production which was generated by the initial two productions, or making two requests to the buffer in one production which could lead to jamming the module.

#### *7.2.4.c Difference between goal and imaginal*

The difference between the two comes down to the use of queries. Because the imaginal module's requests take time one typically needs to make queries to test whether it is free or busy whereas the goal module's requests complete immediately and thus the state is never busy. So, for goal type buffers, production compilation is blocked by any queries in either production because that represents an unusual situation and is thus deemed unsafe. For imaginal type buffers, productions which have queries are allowed when the queries and actions between the productions are "consistent". All of the rules for composition consistency with the imaginal type buffers are a little too complex to describe here, but generally speaking if the first production has a request then the second must either test the buffer for state busy and not also make a request (like motor type buffers) or explicitly test that the buffer is free and make only modifications to the buffer.

For more details, there is a spreadsheet in the docs directory called compilation.xls which contains a matrix for each buffer type indicating what combinations of usages of a buffer between the two productions can be composed.

## 7.3 Utility of newly created productions

So far we have discussed how new production rules are created but not how they are used. When a new production is composed it enters the procedural module and is treated just like the productions which are specified in the model definition. They are matched against the current state along with all the rest of the productions and among all the productions which match the current state the one with the highest utility is selected and fired. When a new production, which we will call **New**, is composed from old productions, which we will call **Old1** and **Old2** and which fired in that order, it is the case that whenever **New** could apply **Old1** could also apply (Note because **New** might be specialized it does not follow that whenever **Old1** could apply **New** could also apply.) The choice between **New**, **Old1**, and any other productions which might also apply will be determined by their utilities as was discussed in the previous unit, and there the utilities were either set in the model with the spp command, or were learned on the basis of rewards.

A newly learned production **New** will initially receive a utility of zero by default (that can be changed with the :nu parameter). Assuming **Old1** has a positive utility value, this means that **New** will almost always lose in conflict resolution with **Old1**. However, each time **New** is recreated from **Old1** and **Old2**, its utility is updated with a reward equal to the current utility of **Old1**, using the same learning equation as discussed in the previous unit:

$U_i(n) = U_i(n-1) + \alpha[R_i(n) - U_i(n-1)]$ **Difference Learning Equation**

As a consequence, even though **New** may not fire initially, its utility will gradually approach the utility of **Old1**. Once the utility of **New** and **Old1** are close enough, **New** will occasionally be selected because of noise. Once **New** is selected itself it will receive a reward like any other production which fires, and its utility can surpass **Old1**'s utility if it is better (it is usually a little better because it typically leads to rewards faster since it saves a production rule firing and often a retrieval from declarative memory).

## 7.4 Learning from Instruction

Generally, production compilation allows a problem to be solved with fewer productions over time and therefore performed faster. In addition to this speed-up, production compilation results in the drop-out of declarative retrieval as part of the task performance. As we saw in the example in the first section, production rules are produced that just "do it" and do not bother retrieving the intervening information. The classic case of where this applies in experimental psychology is in the learning of experimental instructions. These instructions are told to the participant and initially the participant needs to interpret these declarative instructions. However, with practice the participant will come to embed these instructions into productions that directly perform the task. These productions will be like the productions we normally write to model participant performance in the task. Essentially these are productions that participants learn in the warm-up phase of the experiment. The **paired-learning** model for this assignment contains an example of a system that interprets instructions about how to perform a paired associate task and learns productions that do the task directly.

In the model we use the following chunks to represent the understanding of the instructions for the paired associate task (in some of our work we have built productions that read the instructions from the screen and build these chunks but we are skipping that step here to focus on the mechanisms of this unit):

1. (op1 isa operator pre start action read arg1 create post stimulus-read)
2. (op2 isa operator pre stimulus-read action associate arg1 filled arg2 fill post recalled)
3. (op3 isa operator pre recalled action test-arg2 arg1 respond arg2 wait)
4. (op4 isa operator pre respond action type arg2 response post wait)
5. (op5 isa operator pre wait action read arg2 fill post new-trial)
6. (op6 isa operator pre new-trial action complete-task post start)

These are represented as operators that indicate what to do in various states during the course of a paired-associate trial. They consist of a statement of what that state is in the **pre** slot and what state will occur after the action in the **post** slot. In addition, there is an **action** slot to specify the action to perform and two slots, **arg1** and **arg2**, for holding possible arguments needed during the task execution. So to loosely translate the six operators above:

1. At the start read the word and create an encoding of it as the stimulus
2. After reading the stimulus try to retrieve an associate to the stimulus
3. Test whether an item has been recalled and if it has not then just wait
4. If an item has been recalled type it and then wait
5. Store the response you read with the stimulus
6. This trial is complete so start the next one

The model uses a chunk in the **goal** buffer to maintain a current state and sub step within that state and a chunk in the **imaginal** buffer to hold the items relevant to the current state. For this task, the arguments are the stimulus and probe for a trial.

The model must retrieve operators from declarative memory which apply to the current state to determine what to do, and in this simple model we really just need one production rule which requests the retrieval of an operator relevant to the current state:

```
(p retrieve-operator
    =goal>
      isa   task
      state =state
      step  ready
==>
    +retrieval>
      isa   operator
      pre   =state
    =goal>
      step  retrieving-operator)
```

The particular actions specified in the operators (read, associate, test-arg2, type, and complete-task) are all general actions not specific to a paired associate task. We assume that the participant knows how to do these things going into the experiment. This amounts to assuming that there are productions for processing these actions. For instance, the following two productions are responsible for reading an item and creating a chunk in the **imaginal** buffer which encodes the item into the **arg1** slot of that chunk:

```
(p read-arg1
   =goal>
     isa        task
```

```
     step       retrieving-operator
   =retrieval>
     isa        operator
     action     read
     arg1       create
     post       =state
   =visual-location>
     isa        visual-location
   ?visual>
     state      free
   ?imaginal>
     state      free
  ==>
   +imaginal>
     isa        args
     arg1       fill
   +visual>
     cmd        move-attention
     screen-pos =visual-location
   =goal>
     step       attending
     state      =state)

(p encode-arg1
   =goal>
     isa     task
     step    attending
   =visual>
     isa     text
     value   =val
   =imaginal>
     isa     args
     arg1    fill
   ?imaginal>
     state   free
  ==>
   *imaginal>
     arg1    =val
   =goal>
     step    ready)
```

The first production responds to the retrieval of the operator and requests a visual attention shift to an item. It also changes the state in the goal to the operator's post state. The second production modifies the representation in the imaginal buffer with the value from the chunk in the visual buffer and sets the goal to be ready to retrieve the operator relevant to the next state.

The **paired-learning** model responds to the same task as the **paired** model you had for Unit 4. However, rather than having specific productions for doing the task it interprets these operators that represent the instructions for doing this task. For reference, here is the data that is being modeled again:

| Trial | Accuracy | Latency |
|---|---|---|
| 1 | .000 | 0.000 |
| 2 | .526 | 2.156 |
| 3 | .667 | 1.967 |
| 4 | .798 | 1.762 |
| 5 | .887 | 1.680 |
| 6 | .924 | 1.552 |
| 7 | .958 | 1.467 |
| 8 | .954 | 1.402 |

The model can be run either with production compilation on or off. To run it with production compilation off, set the **:epl** parameter to **nil**. The following is a run without production compilation:

```
? (paired-experiment 10)

Latency:
CORRELATION:  0.972
MEAN DEVIATION:  0.192
Trial    1       2       3       4       5       6       7       8
        0.000   2.026   1.970   1.901   1.825   1.777   1.750   1.727

Accuracy:
CORRELATION:  0.991
MEAN DEVIATION:  0.046
Trial    1       2       3       4       5       6       7       8
        0.000   0.398   0.678   0.797   0.868   0.920   0.939   0.948
```

When it is run with **:epl** t, the following is the result:

```
? (paired-experiment 10)

Latency:
CORRELATION:  0.991
MEAN DEVIATION:  0.089
Trial    1       2       3       4       5       6       7       8
        0.000   2.040   1.948   1.831   1.698   1.630   1.584   1.555

Accuracy:
CORRELATION:  0.993
MEAN DEVIATION:  0.046
Trial    1       2       3       4       5       6       7       8
        0.000   0.405   0.642   0.799   0.863   0.899   0.941   0.942
```

As can be seen, whether production compilation is off or on has relatively little effect on the accuracy of recall but turning it on greatly increases the speed-up over trials in recall time. This is because we are cutting out productions and retrievals.

If you set the **:pct** (production compilation trace) parameter to **t** (and you will also need to set **:v** to **t**) you will see the system print out the new productions as they are compiled or the reason why two productions could not be compiled. For instance, the following is a fragment of the trace when we executed the command `(paired-task 1 1)` to study one paired-associate for 1 trial with production compilation turned on.

```
     0.400   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-OPERATOR
Production Compilation process started for RETRIEVE-OPERATOR
  Production ENCODE-ARG1 and RETRIEVE-OPERATOR are being composed.
  New production:

(P PRODUCTION1
  "ENCODE-ARG1 & RETRIEVE-OPERATOR"
   =GOAL>
       STEP ATTENDING
       STATE =STATE
   =IMAGINAL>
       ARG1 FILL
   =VISUAL>
       TEXT T
       VALUE =VAL
   ?IMAGINAL>
       STATE FREE
 ==>
   =GOAL>
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE =STATE
   *IMAGINAL>
       ARG1 =VAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
```

The production that was learned, **Production1**, is a compilation of two of the original productions:

```
(p encode-arg1
   =goal>
     isa     task
     step    attending
   =visual>
     isa     text
     value   =val
   =imaginal>
     isa     args
     arg1    fill
```

```
   ?imaginal>
     state   free
  ==>
   *imaginal>
     arg1    =val
   =goal>
     step    ready)

(p retrieve-operator
   =goal>
     isa     task
     state   =state
     step    ready
  ==>
   +retrieval>
     isa     operator
     pre     =state
   =goal>
     step    retrieving-operator)
```

The first production, **Encode-arg1**, encodes the stimulus, and sets the goal to retrieve the next operator. This is followed by **Retrieve-operator**, which makes the retrieval request. The compiled production is particularly straight forward. Its condition is just the condition of the first production plus the check for the state slot in the goal of the second production, and its action combines the actions of the two.

It is worth understanding how the parameters of this new production are calculated. The value of $\alpha$ is 0.2 (the default). When the production is first created, its utility is set to zero (the default). The utilities of the initial rules in the system are all set to 5 (using the :iu parameter which sets the starting utility for all of the initial productions), so a value of zero means that it will almost certainly lose the competition with the parent production the next time it might be applicable. However, each time it is recreated, it receives a reward which is the same as the utility of the first parent production. Suppose that when the parents fire in sequence again and this same production is recreated, the first parent still has a utility of 5. The utility of the new rule is then updated:

$$\begin{aligned} U_i(n) &= U_i(n-1) + \alpha[R_i(n) - U_i(n-1)] \\ &= 0 + .2(5-0) \\ &= 1 \end{aligned}$$

A utility of 1 is still not sufficient to be selected in competition with a production of utility 5. Thus it will need to be recreated a number of times before it will have a significant chance of being chosen in conflict resolution. The speed of this learning is determined by the setting of $\alpha$. If it is set to 1, productions will typically get very good values immediately and be tried on the first opportunity. If you do that and run (paired-

task 1 10) you will discover in just a few trials it is selecting and firing newly learned productions which are then also going through production compilation:

```
    25.486   PROCEDURAL             PRODUCTION-FIRED PRODUCTION8
Production Compilation process started for PRODUCTION8
  Production RETRIEVE-OPERATOR and PRODUCTION8 are being composed.
  New production:

(P PRODUCTION29
  "RETRIEVE-OPERATOR & PRODUCTION8 - OP6"
   =GOAL>
       STATE NEW-TRIAL
       STEP READY
 ==>
   =GOAL>
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE START
   +GOAL>
       STATE START
       STEP RETRIEVING-OPERATOR
)
```

After just a couple more trials we will start to find productions that are the composition of multiple previously composed productions and it will soon end up with a production like this:

```
(P PRODUCTION52
  "PRODUCTION1 & PRODUCTION31 - OP2"
   =GOAL>
       STATE STIMULUS-READ
       STEP ATTENDING
   =IMAGINAL>
       ARG1 FILL
   =VISUAL>
       TEXT T
       VALUE "zinc"
   ?IMAGINAL>
       STATE FREE
 ==>
   =GOAL>
       STATE RECALLED
       STEP RETRIEVING-OPERATOR
   +RETRIEVAL>
       PRE RECALLED
   *IMAGINAL>
       ARG1 "zinc"
       ARG2 "9"
)
```

Which responds to seeing “zinc” on the screen by placing both “zinc” and “9” in the imaginal buffer chunk’s slots.

## 7.5 Assignment

Your assignment is to make a model that learns the past tense of verbs in English. The learning process of the English past tense is characterized by the so-called U-shaped learning in the learning of irregular verbs. That is, at a certain age children inflect irregular verbs like "to break" correctly, so they say "broke" if they want to use the past tense. But at a later age, they overgeneralize, and start saying "breaked", and then at an even later stage they again inflect irregular verbs correctly. Some people, such as Pinker and Marcus, interpret this as evidence that a rule is learned to create regular past tense (add "ed" to the stem). According to Pinker and Marcus, after this rule has been learned, it is overgeneralized so that it will also produce regularized versions of irregular verbs.

Part of the model is already given in the file past-tense. The assignment is to make a model that learns a production which represents the regular rule for making the past tense and specific productions for producing the past tense of each irregular verb. So eventually it should learn productions which act like this:

IF the goal is to make the past tense of a verb
THEN copy that verb and add –ed

IF the goal is to make the past tense of the verb have
THEN the past tense is had

The code that is provided does two things. It adds correct past tenses to declarative memory, reflecting the fact that a child hears and then encodes correct past tenses in the environment. It also creates goals which indicate to the model that it should generate the past tense of a verb found in the **imaginal** buffer and then runs the model to generate one. The model will be given two correct past tenses for every one that it must generate.

Here are examples of correctly formed past tenses:

```
PAST-TENSE1
      verb have
      stem had
      suffix blank
```

is a correct encoding of the irregular verb have and:

```
PAST-TENSE234
      verb use
      stem use
      suffix ed
```

is a correct encoding of the regular verb use.

At the start of the model's run it will have a chunk in the goal buffer which looks like this:

```
STARTING-GOAL-0
   STATE  START
```

and a chunk in the imaginal buffer which looks like this:

```
CHUNK2-0
   VERB  some-verb
```

where the verb slot holds the verb for which the model must produce a past tense. The model has to fill in the stem and suffix slots of the chunk in the **imaginal** buffer to indicate the past tense form of the verb and then set the state slot of the chunk in the **goal** buffer to done. Once the state slot is set to done, one of the three productions provided with the model should fire to simulate the final encoding and "use" of the word, each of which has a different reward. There are three possible cases:

- An irregular inflection, this is when there is a value in the stem slot and the suffix is marked explicitly as blank. This use has the highest reward, because irregular verbs tend to be short.
- A regular inflection, in which the stem slot is the same as the verb slot and the suffix slot has a value which is not blank. This has a slightly lower reward.
- Non-inflected when neither the stem or suffix slots are set in the chunk. The non-inflection case applies when the model cannot come up with a past tense at all, either because it has no example to retrieve, no production to create it, or no strategy to come up with anything based on a retrieved past tense. The non-inflection solution receives the lowest reward because the past tense would have to be indicated by some other method, for example by adding "yesterday" or some other explicit reference to time.

One important thing to notice is that all three solutions receive a reward. The model receives no feedback as to whether the past tenses it produces are correct – any past tense is considered a success and rewarded. The only feedback it receives is the correctly constructed verbs that it hears from the environment.

You can run the model with the **past-tense** command. It takes as an argument the number of words you want the model to generate:

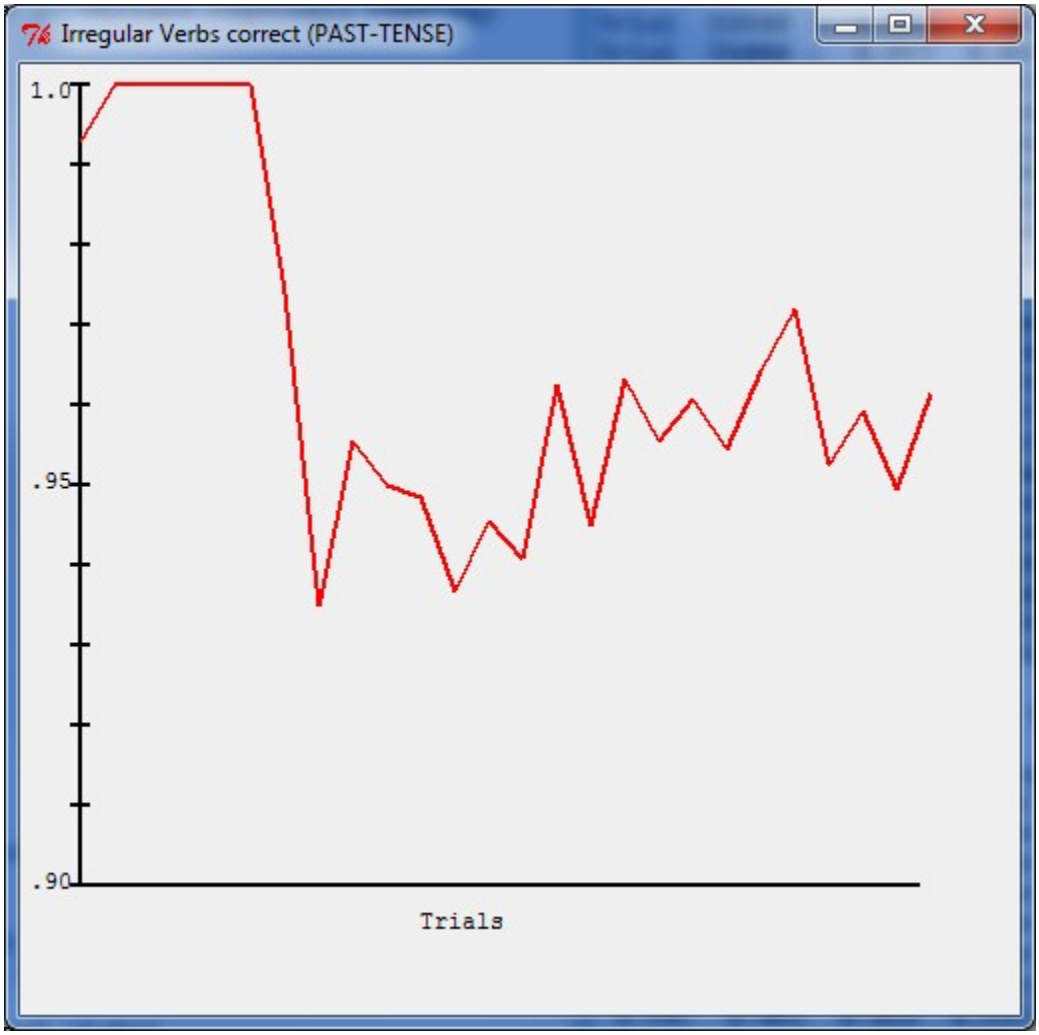
```
(past-tense 5000)
```

As keyword parameters you can specify whether or not you want ACT-R to be verbose (i.e. set the :v parameter to **t**) or whether ACT-R should continue with the run you started in an earlier **past-tense** call. To make the model verbose you would add ":v t" to the call to past-tense and to continue you would add ":cont t". Thus, if you wanted the model to continue running for 1000 more verbs you would call this:

```
(past-tense 1000 :cont t)
```

During the run, the simulation will display four numbers in a row, reflecting the results of the last 100 verbs generated. The first number is the proportion correct of irregular verbs. The second number is the proportion of irregular verbs that are inflected regularly. An increase in this number indicates a regular rule is active i.e. irregular verbs are having "ed" added to them. The third number is the proportion of irregular verbs that are not inflected at all. The fourth number is the proportion of inflected irregular verbs that are inflected correctly (the non-inflected verbs are not counted for this measure). It is in this last column that you should see a U-shape.

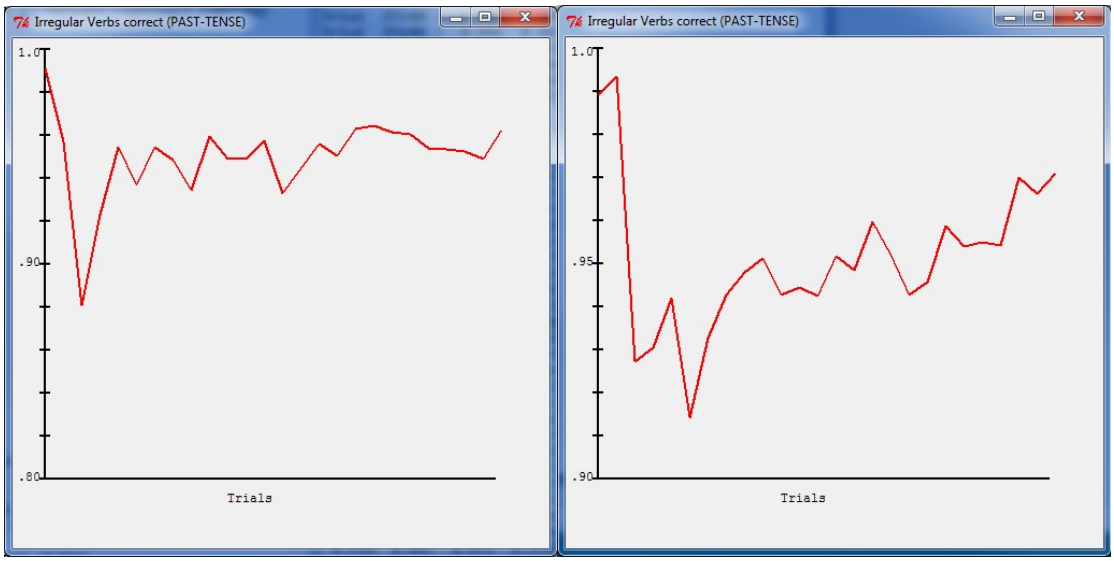
It usually requires more than 5000 trials to see the effect (often 15000~20000 trials are necessary). The :cont (continue) option in the **past-tense** function allows you to run more trials without resetting the model. After the model is done, the **report-irreg** function gives a report where results are summarized for 1000 trials at a time. The 100 trial summaries displayed during the **past-tense** function run are really only there to make sure the model is still doing things, and in what direction the results are going. You can pass t as an optional parameter to the **report-irreg** function to have it generate a graph of the data. What you are looking for from the model is a graph that looks something like this:



It starts out with a high percentage correct, dips down, and then shows an increasing trend (this simplified version of the task will probably not get all the way back to 100% correct). That is the U-shaped learning result. An important thing to look at with the output in the graph is the vertical scale. The graph will adjust the y-intercept to display all of the results and the model should always be getting most of the trials where it does inflect the verb correct (if it's dropping below .75 there's probably a problem with the model).

This model differs from other models in the tutorial in that it does not model a particular experiment, but rather some long term development. This has a couple of consequences for the model. One of those is that using perceptual/motor modules does not contribute much to the objective of the model. Thus things like the "hearing" of past tenses and the eventual generation of the verb in speech are not modeled for the purpose of this exercise. It could be modeled, but it is not what the model and exercise are about. Therefore explicitly adding already processed perceived past tenses to declarative memory and just setting differing rewards for generation of certain classes of verb tenses serves as a reasonable compromise.

The other consequence of the nature of this task is that runs of the model may differ considerably. Here are images of two other runs of the same model as shown above:




On the one hand this is not so bad, as children also differ with respect to U-shaped learning. One reason for the relative unpredictability is the fact that this simulation runs with a very limited vocabulary (extending the vocabulary results in a model that runs extremely slowly which is not practical as an exercise). Another reason is that the effects of noise in the model can have an impact that creates noticeable effects over the long term running of the model and we are only showing the results of a single run – this isn't the average of many simulated children but one child over many simulated weeks of

practice. This also makes comparing this model to data difficult, and hard data on the phenomenon are scarce, although the phenomenon of the U-shape is reported often. A few children have been followed in a longitudinal study, and there is a spreadsheet included with the unit materials (data.xls) that shows those results for comparison.

In terms of the assignment, the objective is to write a model that learns the appropriate productions for producing past tenses. There is no parameter adjustment or data fitting required. All one needs to do is write productions which can generate past tenses based on retrieving previous past tenses, and which through production compilation will over time result in productions which directly apply the regular rule or produce the specific past tense.

The key to a successful model is to implement two different retrieval strategies in the model. The model can either try to remember the past tense for the specific verb or the model can try to generate a past tense based on retrieving any past-tense. These should be competing strategies, and only one applied on any given attempt. If the chosen strategy fails to produce a result the model should "give up" and not inflect the verb. The reason for doing that is that language generation is a rapid process and not something for which a lot of time per word can be allocated.

Additionally, the productions you write should make no explicit reference to either ed or blank because that is what the model is to eventually learn, i.e., you do not write a production that says add ed, but through the production compilation mechanism such a production is learned. Although the experiment code is only outfitted with a limited set of words, the frequency with which the words are presented to the model is in accordance with the frequency they appear in real life, and because of that, if your model learns appropriate productions it should generate the U-shaped learning automatically (although it won't always look the same on every run). Unlike the other models in the tutorial, there is a fairly small set of "good" solutions to this task which will result in the generation of the U-shaped learning because the model's starting productions will need to result in the learning of specific productions over time to work correctly. One final thing to note is that even a "correct" model may sometimes fail to show the U-shape, but on most runs (90% or more) it should show up in some form.

There are two important things to make sure to address when writing your model. It must set the state slot of the chunk in the goal buffer to done on each trial, and the chunk in the imaginal buffer must have one of the forms described above: either a chunk with values in each of the verb, stem and suffix slots or a chunk with only a value in the verb slot. Those conditions are necessary so that one of the provided productions will fire and propagate a reward. If it does not do so, then there will be no reward propagated to promote the utility learning for that trial and the reward from a later trial will be propagated back to the productions which fired on the trial which didn't get a reward. That later reward will be very negative because there are 200 seconds between trials, and that will make it very difficult, if not impossible, to produce the U-shaped learning. Essentially this represents the model saying something every time it tries to produce a past-tense while speaking.

One final thing to note is that when the model doesn't give up it should produce a reasonable answer. One aspect of being reasonable is that the resulting chunk in the imaginal buffer should have the same value in the verb slot that it started with. Similarly, if there is a value in the stem slot, then it should be either that verb or the correct irregular form of that verb. For example, if it starts with "have" in the verb slot acceptable values would be "have" or "had" in the stem slot, but if it starts with a regular verb, like "use" the only acceptable value for the stem would be "use" because it shouldn't be trying to create some irregular form for a regular verb. If the model does produce an unreasonable result there will be a warning printed showing the starting verb and the result which the model created like this:

```
#|Warning: Incorrectly formed verb.  Presented CALL and produced
VERB GET STEM GOT SUFFIX BLANK. |#
```

That indicates the model was asked to produce the past tense for "call" and responded with the correct past tense for the different verb "get". Your model should not produce any incorrectly formed verb warnings when it runs.