Procedural Learning Modeling Issues

This text will cover some of the issues that can arise when working with utility learning and production compilation and describe ways that the Environment tools may be used to help. Models which use the procedural learning mechanisms often do so in conjunction with declarative learning and also often require running for many trials for the learning effects to show up. Because of that, such models can be more difficult to analyze and debug since one may need to investigate both declarative and procedural issues over long runs. To better demonstrate things for this text we will be using two simple tasks which are focused only on the procedural issues involved. The mechanisms described here can be used in conjunction with those described previously for declarative memory, and we will also indicate ways of dealing with longer runs.

Utility Learning

First we will look at a model which is using utility learning in a task similar to the choice experiment from unit 6 found in the “utility-learning-issues.lisp” file. The code for the task can be found in the ul-issues.lisp and ul_issues.py files. We do not have any experimental data which we will be fitting with this model, but we do have an expectation that it will learn which choice is better. That learning should show up as a higher utility for the production which chooses the better response and we will look for that as the model runs.

The Task

In this task the model must choose one of two options, either A or B, within five seconds. Then after the five seconds have passed the model will either be presented with the correct response for this trial or informed that no answer will be provided for the trial. The feedback will be presented for two seconds and then the next trial will begin. Thus, each trial lasts exactly seven seconds. For this task, choice A will be reported as correct on 60% of the trials, 20% of the trials will indicate choice B as correct, and 20% of the trials will provide no feedback. Thus, we will expect the model to learn to choose option A more frequently than option B.

Because this task only has to run with the model it has been implemented by directly manipulating the chunks in the model’s **goal** and **imaginal** buffers. The task will put a chunk in the **goal** buffer indicating that it is time to choose and then put a chunk in the **imaginal** buffer with the feedback five seconds later. Two seconds after that it will provide another **goal** chunk indicating the next choice time, and that process will repeat for as long as the model runs. The task operates by scheduling the actions to occur for the model, and does not require calling a function other than run. There is no data collected or results reported for the task, but the choice and feedback actions will be shown in the medium detail trace for reference. Because the **imaginal** buffer tests a slot that is set from code outside of the model we have used declare-buffer-usage to avoid the style warnings about that slot’s usage in the productions.

The Model

Because the task is directly modifying the chunks in the buffers, the model can simply consist of five productions. Two productions respond to the **goal** buffer chunk indicating that it is time to choose, one for each choice, and there are three productions which process the feedback provided in the **imaginal** buffer. The feedback handling productions consist of one which fires when the model chose correctly, one which fires when it chose incorrectly, and one which fires when there is no feedback for the trial. We will not show the productions here, but there is nothing new or unusual about them so they should be easy to understand by looking at the model file. The only learning mechanism enabled in the model is utility learning and the model has been given some noise in utilities with these parameter settings:

```
(sgp :esc t :ul t :egs .5)
```

The utility learning rate parameter :alpha is not set so it will have the default value of .2. To allow the model to learn, the productions which fire for matching and mismatching feedback are given the following rewards:

```
  (spp response-matches :reward 4)
  (spp response-doesnt-match :reward 0)
```

The productions are not given any particular starting utilities. Therefore, they will all start with the default utility of 0.

The last line of the model definition schedules the first choose event to happen at time 0, and that starts the cycle of scheduling the events to drive the task as the model runs.

Testing the Model

Loading the ul-issues.lisp file or importing the ul_issues.py file will automatically load the model file, and when we do so there are no warnings or errors reported so we can start running it now. One thing that we could do would be to just run it for several trials and then see how the utilities have changed by that point. If the choose-a production has a higher utility than choose-b we might then consider the model done. However, as has been mentioned in the other testing texts, it is always better to start small and make sure to understand how the model is working and learning before moving on to look at the higher level results.

As a first test we should run a couple of trials and make sure the model is operating as we would expect. Here is the trace from the first trial after running for just under seven seconds:

```
     0.000   NONE                   utility-learning-issues-choose
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     5.000   NONE                   utility-learning-issues-show-result a
     5.000   PROCEDURAL             CONFLICT-RESOLUTION
     5.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-MATCHES
     5.050   PROCEDURAL             CLEAR-BUFFER GOAL
     5.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     5.050   UTILITY                PROPAGATE-REWARD 4
     5.050   PROCEDURAL             CONFLICT-RESOLUTION
     6.950   ------                 Stopped because time limit reached
```

We see the model choose A, the feedback presented is that A is the correct choice, then the model fires the response-matches production and a reward of 4 is applied. That looks good, but we should check a couple more trials to make sure. Here is the trace for the next two:

```
     7.000   NONE                   utility-learning-issues-choose
     7.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     7.000   PROCEDURAL             CONFLICT-RESOLUTION
     7.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     7.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.000   NONE                   utility-learning-issues-show-result NIL
    12.000   PROCEDURAL             CONFLICT-RESOLUTION
    12.050   PROCEDURAL             PRODUCTION-FIRED UNKNOWN-RESPONSE
    12.050   PROCEDURAL             CLEAR-BUFFER GOAL
    12.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    12.050   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   NONE                   utility-learning-issues-choose
    14.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    14.000   PROCEDURAL             CONFLICT-RESOLUTION
    14.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-B
    14.050   PROCEDURAL             CONFLICT-RESOLUTION
    19.000   NONE                   utility-learning-issues-show-result a
    19.000   PROCEDURAL             CONFLICT-RESOLUTION
    19.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-DOESNT-MATCH
    19.050   PROCEDURAL             CLEAR-BUFFER GOAL
    19.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    19.050   UTILITY                PROPAGATE-REWARD 0
    19.050   PROCEDURAL             CONFLICT-RESOLUTION
    20.950   ------                 Stopped because time limit reached
```

There we see trials with the model responding to both the lack of feedback and a trial when it responds incorrectly. Since that all looks good we should now look at how the utility learning is progressing as it runs.

Because this is a small model it should be easy to follow the learning by simply enabling the utility learning trace and watching the values change. If it were a larger model however that might not be as tractable, and we might need to use some of the Environment tools to help as will be discussed later. For now, we will just enable the utility learning trace by turning it on after resetting the model using sgp in Lisp or set_parameter_value in Python:

```
? (sgp :ult t)
```

```
>>> actr.set_parameter_value(':ult',True)
```

When we run it now we see the update to utility for the first reward:

```
     0.000   NONE                   utility-learning-issues-choose
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     5.000   NONE                   utility-learning-issues-show-result a
     5.000   PROCEDURAL             CONFLICT-RESOLUTION
     5.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-MATCHES
     5.050   PROCEDURAL             CLEAR-BUFFER GOAL
     5.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     5.050   UTILITY                PROPAGATE-REWARD 4
 Utility updates with Reward = 4.0   alpha = 0.2
```

```
  Updating utility of production CHOOSE-A
   U(n-1) = 0.0   R(n) = -1.0500002 [4.0 - 5.05 seconds since selection]
   U(n) = -0.21000004
  Updating utility of production RESPONSE-MATCHES
   U(n-1) = 0.0   R(n) = 3.95 [4.0 - 0.05 seconds since selection]
   U(n) = 0.79
```

Looking at the change in utility for the production choose-a on this trial indicates that there seems to be a problem. The model chose A and the feedback provided indicated that A was the correct choice which lead to a positive reward, but the utility of the choose-a production decreased from 0 to -0.21. The reason for that is because the effective reward a production receives is discounted by the time that passed between the production's selection and when the reward is received. In this task there are 5.05 seconds between the choice and the reward. Thus, with a reward of 4 being provided on a correct response we end up penalizing the production.

Generally, that is not a good situation since it means that the model would be less likely to choose A after positive feedback. If we want the reward to have a positive effect then we should make sure that it is large enough to do so considering the amount of time that passes. To fix that for this model we will adjust the reward provided for being correct to 6 instead of 4:

```
  (spp response-matches :reward 6)
```

While we are editing the model file we should also add the :ult setting so that we don't have to keep setting it each time we reset to start a new run:

```
  (sgp :esc t :ul t :egs .5 :ult t)
```

After making that change and saving the model here is the trace of the utility learning now:

```
     0.000   NONE                   utility-learning-issues-choose
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
     0.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
     5.000   NONE                   utility-learning-issues-show-result a
     5.000   PROCEDURAL             CONFLICT-RESOLUTION
     5.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-MATCHES
     5.050   PROCEDURAL             CLEAR-BUFFER GOAL
     5.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
     5.050   UTILITY                PROPAGATE-REWARD 6
 Utility updates with Reward = 6.0   alpha = 0.2
  Updating utility of production CHOOSE-A
   U(n-1) = 0.0   R(n) = 0.9499998 [6.0 - 5.05 seconds since selection]
   U(n) = 0.18999997
  Updating utility of production RESPONSE-MATCHES
   U(n-1) = 0.0   R(n) = 5.95 [6.0 - 0.05 seconds since selection]
   U(n) = 1.1899999
```

We now see a positive change in the choose-a production's utility after choosing it correctly on this trial.

Had we just been looking at the model's performance over a long run we may not have noticed this oddity in the model's learning pattern. For example, had we just run the model for 100 trials from the initial state and looked at the resulting utilities we would have seen something like this if we use spp to print out the results:

```
? (spp choose-a choose-b)

>>> actr.spp('choose-a','choose-b')
```

```
Parameters for production CHOOSE-A:
 :utility -4.107
 :u  -4.405
 :at  0.050
 :reward    NIL
Parameters for production CHOOSE-B:
 :utility -5.604
 :u  -5.618
 :at  0.050
 :reward    NIL
(CHOOSE-A CHOOSE-B)
```

Production choose-a has a higher utility than choose-b which means that the model will be choosing A more often than B. So, even with a successful choice penalizing the model initially, in the long term the model still gets to the expected result since presumably the incorrect trials are penalized even more, but if we are concerned with how it gets there, which often is the reason for creating a learning model, we should pay attention to the details along the way. In this case, the negative utilities may have been an indication that there was a problem, but if instead of looking at the utilities we had been looking at response data like choice percentages we may not have noticed at all.

Now that we have the first trial operating in a reasonable manner we will look at the next trial. Here is the trace for the second trial:

```
     7.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     7.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.000   NONE                   utility-learning-issues-show-result NIL
    12.000   PROCEDURAL             CONFLICT-RESOLUTION
    12.050   PROCEDURAL             PRODUCTION-FIRED UNKNOWN-RESPONSE
    12.050   PROCEDURAL             CLEAR-BUFFER GOAL
    12.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    12.050   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   NONE                   utility-learning-issues-choose
    14.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    14.000   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   ------                 Stopped because time limit reached
```

The model chooses A again but this time there is no feedback. Because the unknown-response production provides no reward there is no change to the utility of the choose-a production. So we will now look at the next trial:

```
    14.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-B
    14.050   PROCEDURAL             CONFLICT-RESOLUTION
    19.000   NONE                   utility-learning-issues-show-result a
    19.000   PROCEDURAL             CONFLICT-RESOLUTION
    19.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-DOESNT-MATCH
    19.050   PROCEDURAL             CLEAR-BUFFER GOAL
    19.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    19.050   UTILITY                PROPAGATE-REWARD 0
```

```
 Utility updates with Reward = 0.0   alpha = 0.2
  Updating utility of production CHOOSE-A
   U(n-1) = 0.18999997   R(n) = -12.05 [0.0 - 12.05 seconds since selection]
   U(n) = -2.258
  Updating utility of production UNKNOWN-RESPONSE
   U(n-1) = 0.0   R(n) = -7.05 [0.0 - 7.05 seconds since selection]
   U(n) = -1.4100001
  Updating utility of production CHOOSE-B
   U(n-1) = 0.0   R(n) = -5.05 [0.0 - 5.05 seconds since selection]
   U(n) = -1.0100001
  Updating utility of production RESPONSE-DOESNT-MATCH
   U(n-1) = 0.0   R(n) = -0.05 [0.0 - 0.05 seconds since selection]
   U(n) = -0.010000001
    19.050   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   NONE                   utility-learning-issues-choose
    21.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    21.000   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   ------                 Stopped because time limit reached
```

This time the model chooses B, but the feedback indicates that A was the correct choice. The response-doesnt-match production fires and provides a reward of 0 which gets propagated back. However, in addition to penalizing the choose-b production as we would expect it also penalizes the choose-a production. That happens because the reward affects all productions which have fired since the previous reward, which occurred after response-matches fired on the first trial. Because there was no reward provided on the second trial when unknown-response fired this reward gets applied to the productions for that trial as well.

To prevent that from happening we will have to provide a reward on the trials without any feedback when unknown-response fires. The question becomes how much reward should we provide when there is no feedback? As always, there is no single answer to such a question and depending on the task and hypothesis behind the model, values anywhere between the positive and negative feedback may be appropriate. Alternatively, instead of picking a value, there is a special option available for the reward which we will describe and use here.

The utility learning mechanism provides the option of specifying a “null reward”. Such a reward does not adjust the utilities of any productions, but it does cause the marker for when the last reward was provided to be updated. That allows the modeler to indicate that there was nothing to be learned since the last reward was provided. As with choosing which reward values to provide, the modeler will have to decide if a null reward value is appropriate for any particular situation.

Any non-numeric true value provided as a reward results in a null reward for the model. If one is providing rewards to the model automatically with the firing of productions, as is done in this example, then setting the reward value for a production to t instead of a number is how one specifies the null reward. If instead one is using the trigger-reward command to provide rewards to the model directly then any true non-numeric value can be provided to produce the null reward. The reason for allowing any value when using trigger-reward is because it will show in the trace and will be passed to monitoring commands which may be helpful when looking at the trace or developing the task code.

Here is the setting we will add to the model to provide a null reward when there is no feedback for a trial:

```
(spp unknown-response :reward t)
```

By providing a null reward when the unknown-response production fires it will stop the reward from the next trial from propagating back past that point.

After saving that change in the model and reloading it here is what we see now for the utility update on the second and third trials:

```
     7.000   PROCEDURAL             CONFLICT-RESOLUTION
     7.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-A
     7.050   PROCEDURAL             CONFLICT-RESOLUTION
    12.000   NONE                   utility-learning-issues-show-result NIL
    12.000   PROCEDURAL             CONFLICT-RESOLUTION
    12.050   PROCEDURAL             PRODUCTION-FIRED UNKNOWN-RESPONSE
    12.050   PROCEDURAL             CLEAR-BUFFER GOAL
    12.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    12.050   UTILITY                PROPAGATE-REWARD T
  Non-numeric reward clears utility learning history.
    12.050   PROCEDURAL             CONFLICT-RESOLUTION
    14.000   NONE                   utility-learning-issues-choose
    14.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    14.000   PROCEDURAL             CONFLICT-RESOLUTION
    14.050   PROCEDURAL             PRODUCTION-FIRED CHOOSE-B
    14.050   PROCEDURAL             CONFLICT-RESOLUTION
    19.000   NONE                   utility-learning-issues-show-result a
    19.000   PROCEDURAL             CONFLICT-RESOLUTION
    19.050   PROCEDURAL             PRODUCTION-FIRED RESPONSE-DOESNT-MATCH
    19.050   PROCEDURAL             CLEAR-BUFFER GOAL
    19.050   PROCEDURAL             CLEAR-BUFFER IMAGINAL
    19.050   UTILITY                PROPAGATE-REWARD 0
 Utility updates with Reward = 0.0   alpha = 0.2
  Updating utility of production CHOOSE-B
   U(n-1) = 0.0   R(n) = -5.05 [0.0 - 5.05 seconds since selection]
   U(n) = -1.0100001
  Updating utility of production RESPONSE-DOESNT-MATCH
   U(n-1) = 0.0   R(n) = -0.05 [0.0 - 0.05 seconds since selection]
   U(n) = -0.010000001
    19.050   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   NONE                   utility-learning-issues-choose
    21.000   GOAL                   SET-BUFFER-CHUNK GOAL INITIAL-GOAL NIL
    21.000   PROCEDURAL             CONFLICT-RESOLUTION
    21.000   ------                 Stopped because time limit reached
```

On the second trial it now reports that there is a null reward which clears the history and sets a new marker for the last reward given. Then on the third trial only the choose-b and response-doesnt-match productions get an update to their utilities.

After that change the model seems to be working as we would expect now – it gets a positive reward for guessing correctly, no change to rewards when there is no feedback, and a negative reward when it guesses incorrectly. If we check the utility values of the choose-a and choose-b productions now we see that the :u value for choose-a is greater than the :u value for choose-b:

```
Parameters for production CHOOSE-A:
 :utility  1.111
 :u   0.190
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production CHOOSE-B:
 :utility -1.961
 :u  -1.010
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

As a test, we can run the model for several more trials and look at the results, and we can turn off the trace using with-parameters, sgp, or the set_parameter_values functions so it runs faster, and then look at the parameter values:

```
Parameters for production CHOOSE-A:
 :utility -1.665
 :u  -0.646
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
Parameters for production CHOOSE-B:
 :utility -3.326
 :u  -2.982
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
```

We see that choose-a still has the greater U(n) value, though both are negative. Before considering why they are both negative, we will compute the probability that the model will fire choose-a instead of choose-b at this time using the equation from unit 6 of the tutorial:

$$\Pr obability(choose-a) = \frac{e^{\frac{-0.646}{.5*\sqrt{2}}}}{e^{\frac{-0.646}{.5*\sqrt{2}}} + e^{\frac{-2.982}{.5*\sqrt{2}}}} \approx .965$$

That is likely a little higher than we would want if we were trying to fit human performance, but without any explicit data to fit we will not adjust that in this model.

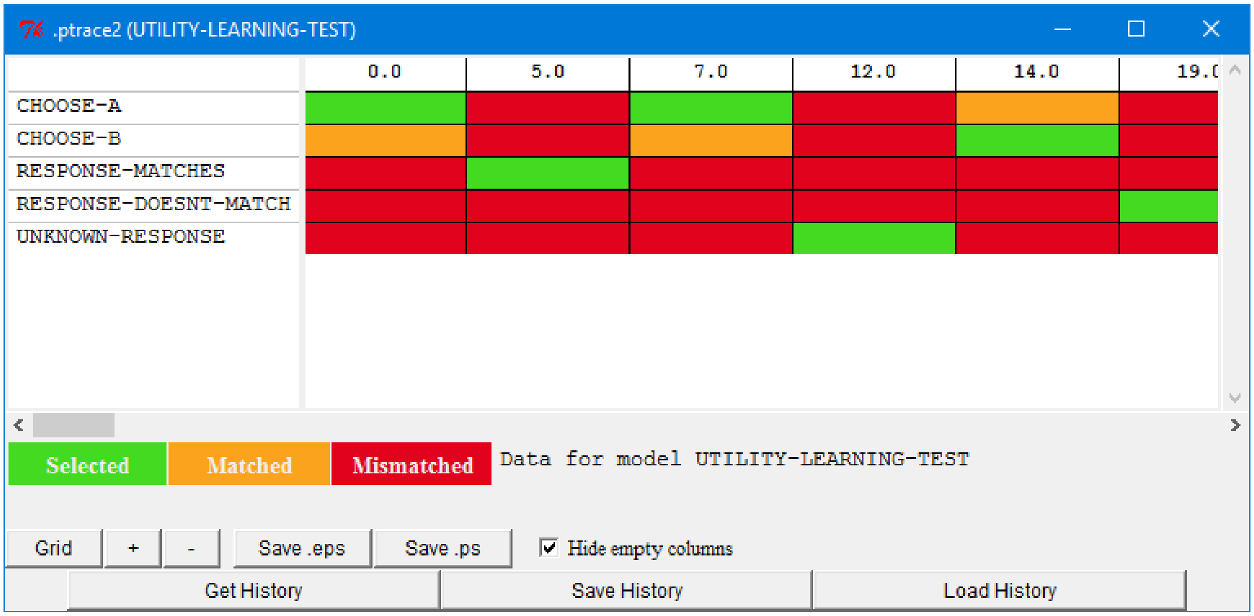
Now, as for why the values are negative, if we look back at the traces we will see that the penalty for an incorrect response is -5.05 whereas the benefit for a correct response is only +0.95. That much larger penalty for being incorrect appears to be what is driving the values negative, but we will look into that further below to make sure there is not some other issue. Because the utility values are only meaningful in comparison among competing productions, having negative values is not in and of itself a bad thing that always needs to be corrected. One situation where that might be an issue however is if one is using production compilation and has left the default utility value for newly learned productions at 0. If the original productions have negative utilities then the newly learned productions with utilities of 0 will be immediately more likely to be selected. That situation is not recommended and one would likely want to adjust the starting utilities of the original productions, adjust the initial utility for new productions, or adjust the rewards that are provided so that a more gradual introduction of the newly learned productions occurs. Since this model is not using production compilation, as long as we do not find something wrong with how it is operating we will not attempt to adjust the rewards or other parameters to eliminate the negative utilities.

Using the utility trace to investigate the changes to the utility of the productions works alright when dealing with a few trials in a small model, but if the task requires lots of trials or has lots of competing productions then reading through the trace can be difficult and time consuming. The "Production" tools in the Environment, which were introduced in the unit 3 modeling text, may help to investigate utility issues for longer runs and we will look at doing so below. Using those tools may not always explain what has happened, but when they do not they should at least help to find where problems are occurring so that a more detailed investigation can be done using more fine grain tools.

With the production grid we can get an overview of which productions are competing and which one, if any, is selected. As described in the unit 3 text, to use the tool we should open it before running the model so that it can record the data needed. When working with longer runs it can also help to have the tool hide the empty columns. That can be done by checking the "Hide empty columns" box near the bottom of the window. We will turn off the trace in the model definition for now since we will be looking at the information through the Environment:

```
(sgp :esc t :ul t :egs .5 :ult t :v nil)
```

We will save that change, reload the model, and then open the "Production" grid tool so that it records the data. We will run it for 40 trials and then press the "Get history" button to see the data. That should result in a display which looks something like this:

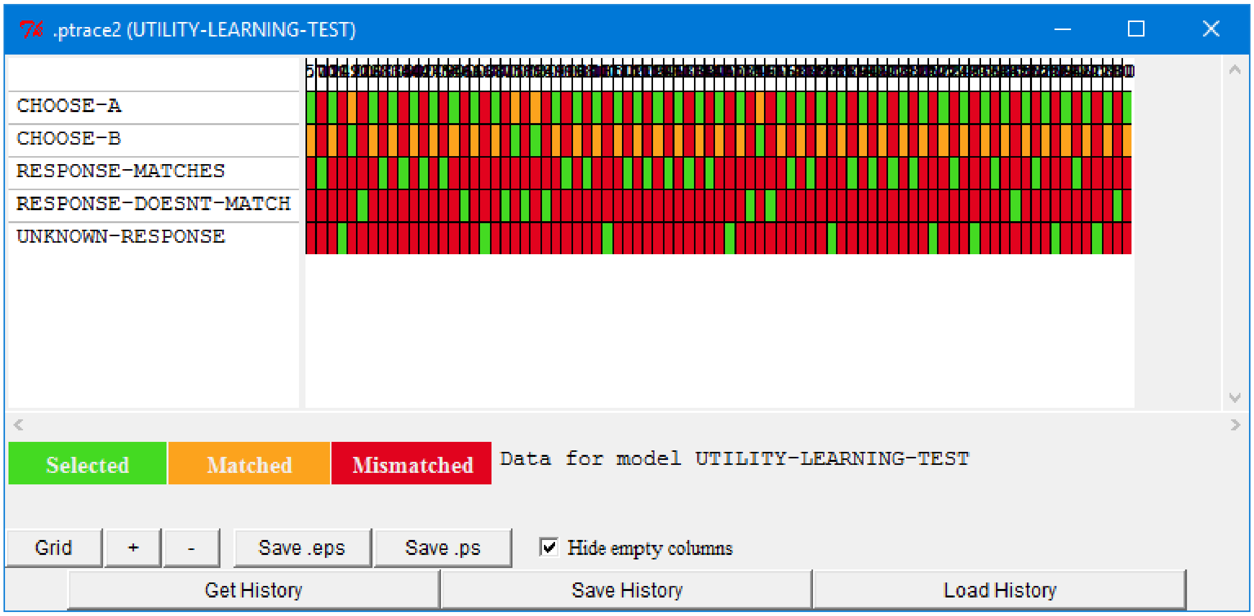


	0.0	5.0	7.0	12.0	14.0	19.
CHOOSE-A	🟩	🟥	🟩	🟥	🟧	🟥
CHOOSE-B	🟧	🟥	🟧	🟥	🟩	🟥
RESPONSE-MATCHES	🟥	🟩	🟥	🟥	🟥	🟥
RESPONSE-DOESNT-MATCH	🟥	🟥	🟥	🟥	🟥	🟩
UNKNOWN-RESPONSE	🟥	🟥	🟥	🟩	🟥	🟥

🟩 Selected 🟧 Matched 🟥 Mismatched

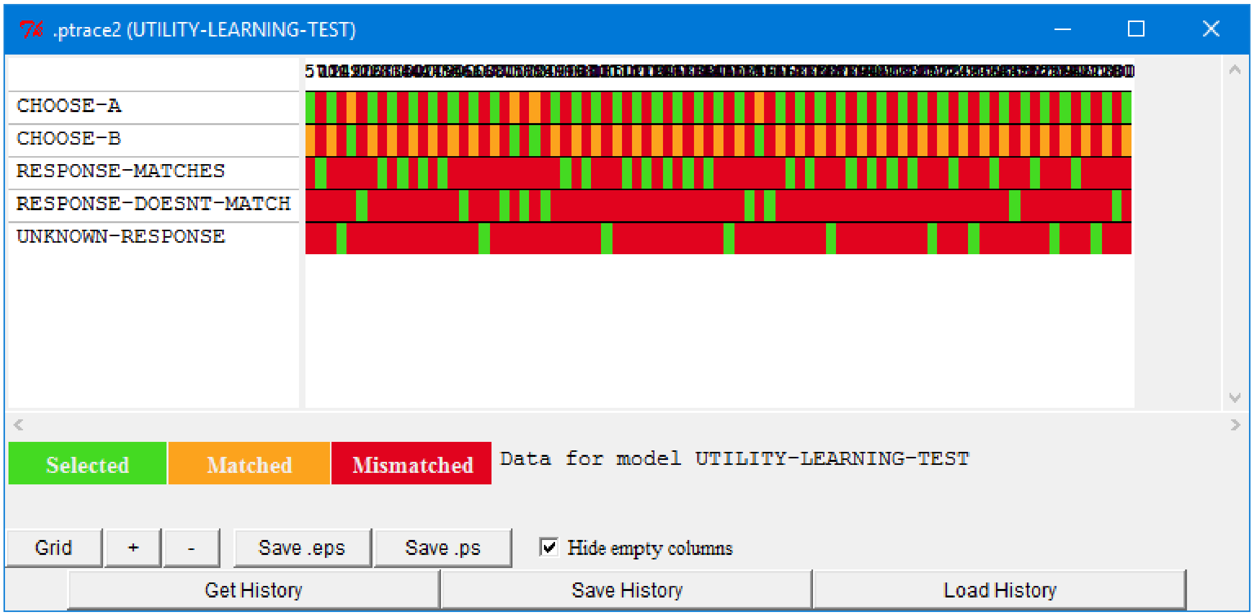
Data for model UTILITY-LEARNING-TEST

Each column is a conflict-resolution event. The green and orange productions indicate which ones matched the current state and the green one is the one that was selected. Above we can see the first three trials where the model chose A the first two times and then B on the third one. We could scroll the view horizontally to see all the trials, but looking at the whole sequence at once can often be more informative. To do that, we need to zoom out the display by pressing the "-" button at the bottom of the window. After pressing that a few times we can have the entire 40 trial sequence visible at once and that will look like this:



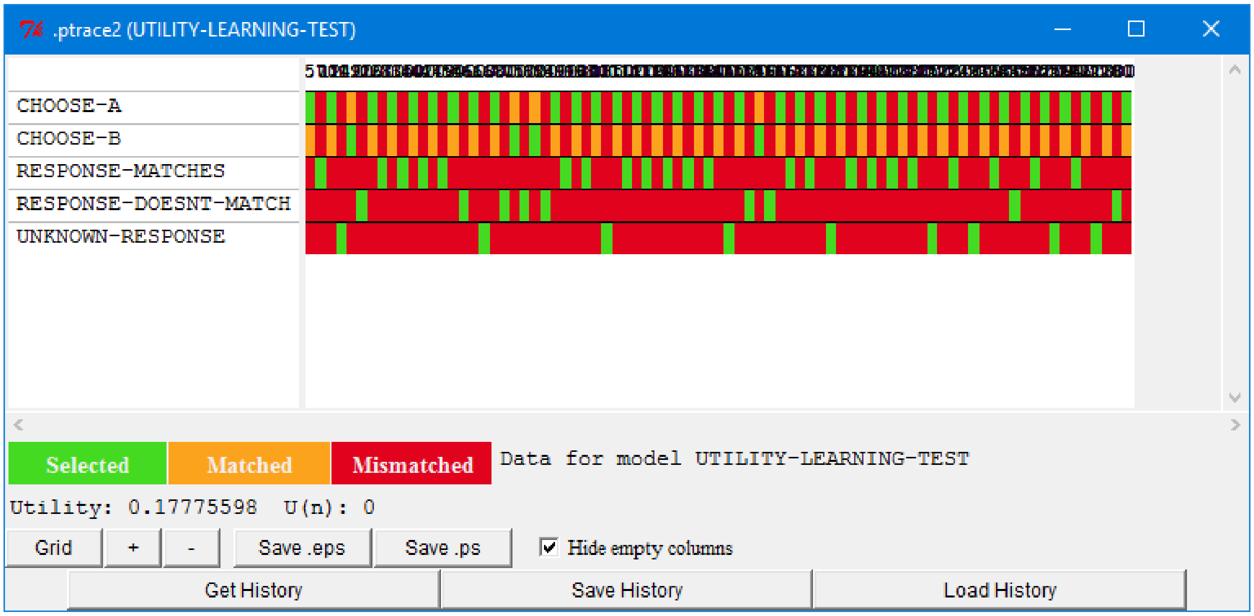
Data for model UTILITY-LEARNING-TEST

You may not always want to zoom out that far, but for purposes of this example we will look at the entire run. One thing that can help when zooming out with this tool is to turn off the display of the black likes separating the columns. To do that you can press the "Grid" button at the bottom and then the display will look like this which may be a little easier to look at:



Data for model UTILITY-LEARNING-TEST

Looking at that display we can see that choose-a gets selected a lot more often than choose-b which is what we expected from the model. If we are interested in the utility values at particular times we can also see those by placing the mouse cursor over the green or orange bars in the display. Here is what it shows for the first green bar in the choose-a row:

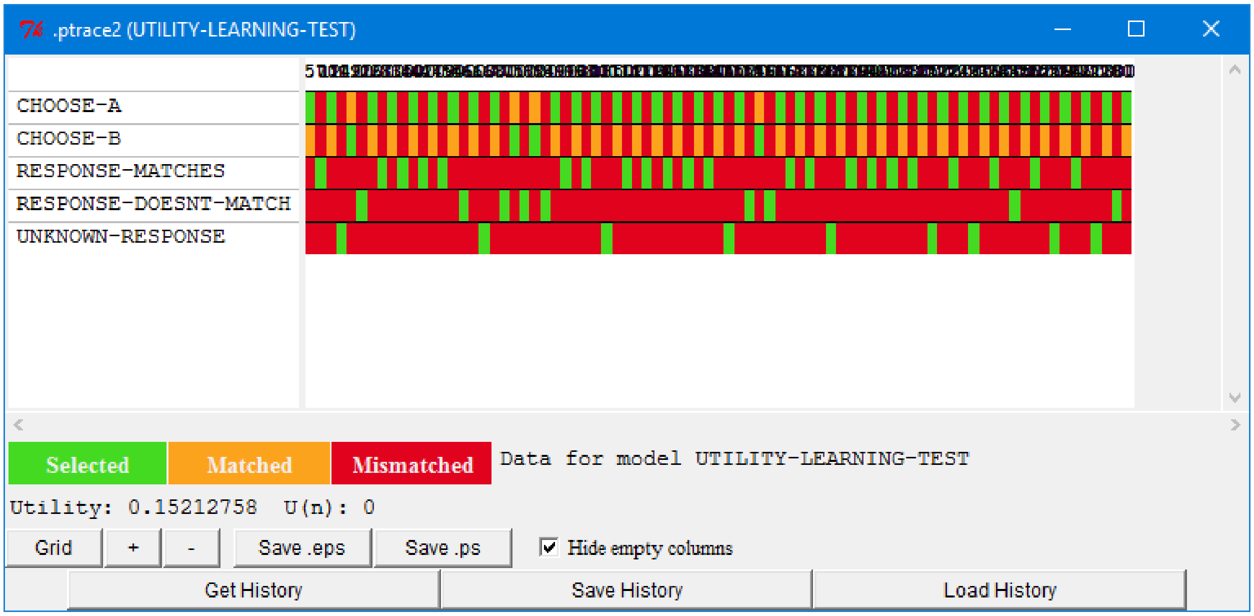


```
CHOOSE-A
CHOOSE-B
RESPONSE-MATCHES
RESPONSE-DOESNT-MATCH
UNKNOWN-RESPONSE
```

☑ Hide empty columns

```
Data for model UTILITY-LEARNING-TEST
Utility: 0.17775598  U(n): 0
```

It shows the noisy utility value which was used during that conflict-resolution action and the true U(n) value for the production at that time. In this case the U(n) is 0 since that is before any rewards have been applied. If we look at the first choose-b occurrence (the orange box) we see that it also has a U(n) of 0 and its utility was less than the utility of choose-a which is why choose-a was chosen at that time:



```
CHOOSE-A
CHOOSE-B
RESPONSE-MATCHES
RESPONSE-DOESNT-MATCH
UNKNOWN-RESPONSE
```

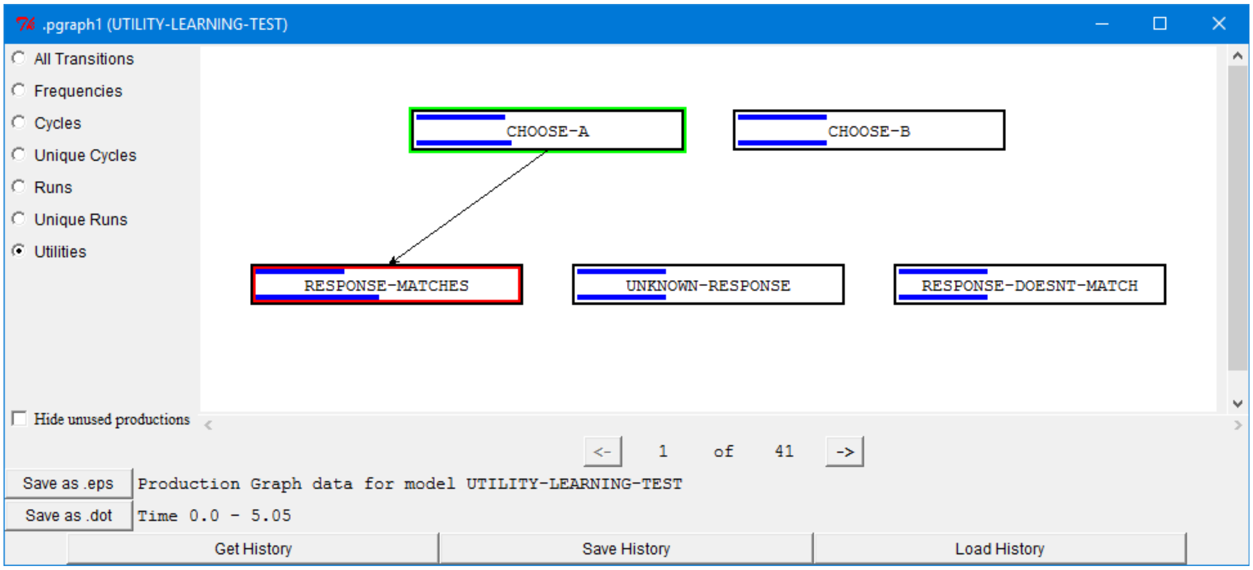
☑ Hide empty columns

```
Data for model UTILITY-LEARNING-TEST
Utility: 0.15212758  U(n): 0
```

Using this tool we could look at the utility change for each trial to see how things are changing from trial to trial and you should feel free to investigate that. However, we will not be walking though that in this text. Instead, we will look at an alternative way to view that information using the “Production” graph tool.

The “Production” graph tool can display the sequence of production firings broken into segments based on when the model received rewards and in that view it will also show the utility changes which occurred. The “Production” graph tool relies on the same recorded data as the “Production” grid tool so since we have been using that tool the information is already available so we can just open the “Production” graph tool now and view the data without having to run the model again.

After opening a production graph display, to get the information we are interested in we need to select the “Utilities” option on the left and then press the “Get History” button. That will result in a display which looks like this after adjusting the window size to see all of the boxes:



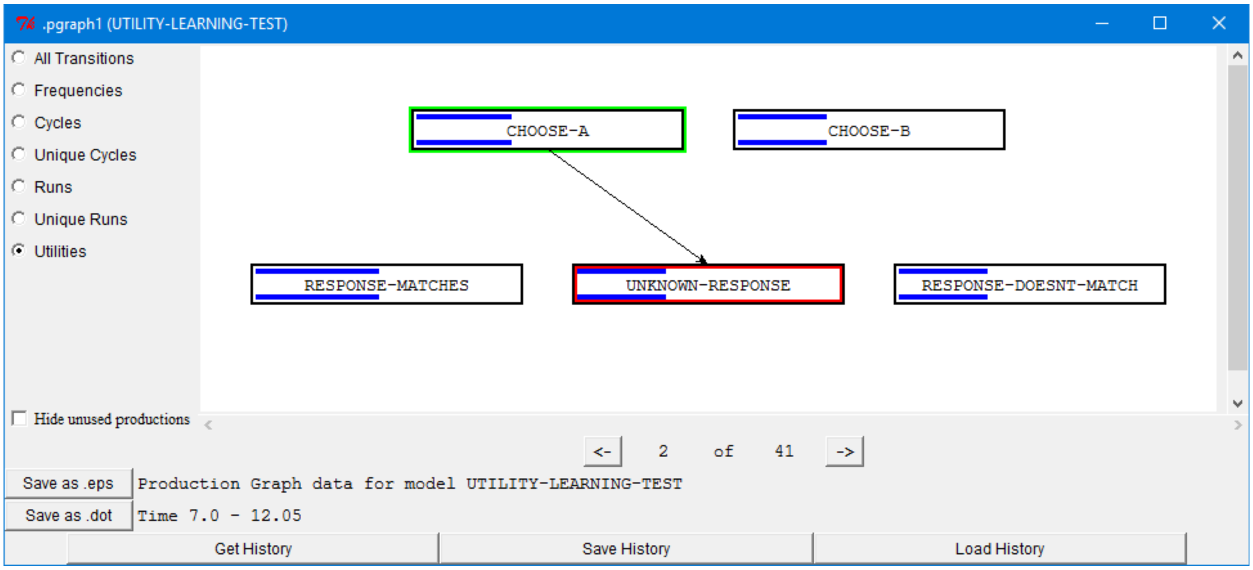
```
Production Graph data for model UTILITY-LEARNING-TEST
Time 0.0 - 5.05
```

The display is similar to the one shown earlier in the tutorial: it displays the sequence of productions which fired as a directed graph starting at the production highlighted in green and ending with a production highlighted in red. The “Utilities” display however differs in a couple of ways from those seen previously. The first is that now the run is broken up into separate graphs based on when the model receives rewards. The red highlighted productions will be the last production to fire before a reward is received (except for the last display where it might be just the final production which the model has fired whether or not it is followed by a reward). Since we have a reward provided on each trial in this task there will be one graph for each trial of this run, but the display shows that there are 41 total graphs to view. That is because the model has already selected a production at the start of the 41st trail which results in another graph to be displayed. The other difference from the previous production graph displays is that now in each production’s box we see two blue lines. The one at the top represents the true utility of the production before the reward was provided and the one below represents the true utility after the reward has been propagated. The bars start at the left of the box and increase in length with the utility value. All of the productions are displayed in boxes of the same width and the utilities are scaled across all of the productions and graphs. A blue bar of length zero represents the minimum utility value that any production has across the entire run and a bar the width of the production box will be the maximum utility that occurs for any production over the entire run.

While this display does not show the actual values of the utilities, the relative changes that it does show should be sufficient to verify that things are working as desired and should be easier to go through than reading all the utility trace information. We will only show a couple of the graphs here for reference, but you may want to step through all of them on your own to make sure you understand how the model operates.

Looking at the display above we see that the model chose A on the first trail and that the response-matches production fires indicating a correct choice. When that happens we see that both the choose-a and response-matches productions had their utilities increased while the others stayed the same (which we also saw earlier in the utility trace). By using the graphic display it should be easier to look at the changes that occur on each trial than it would be to read through all of the utility traces. We will only show a couple more examples from this run below, but you may want to look at the whole sequence to verify for yourself that it works as expected.

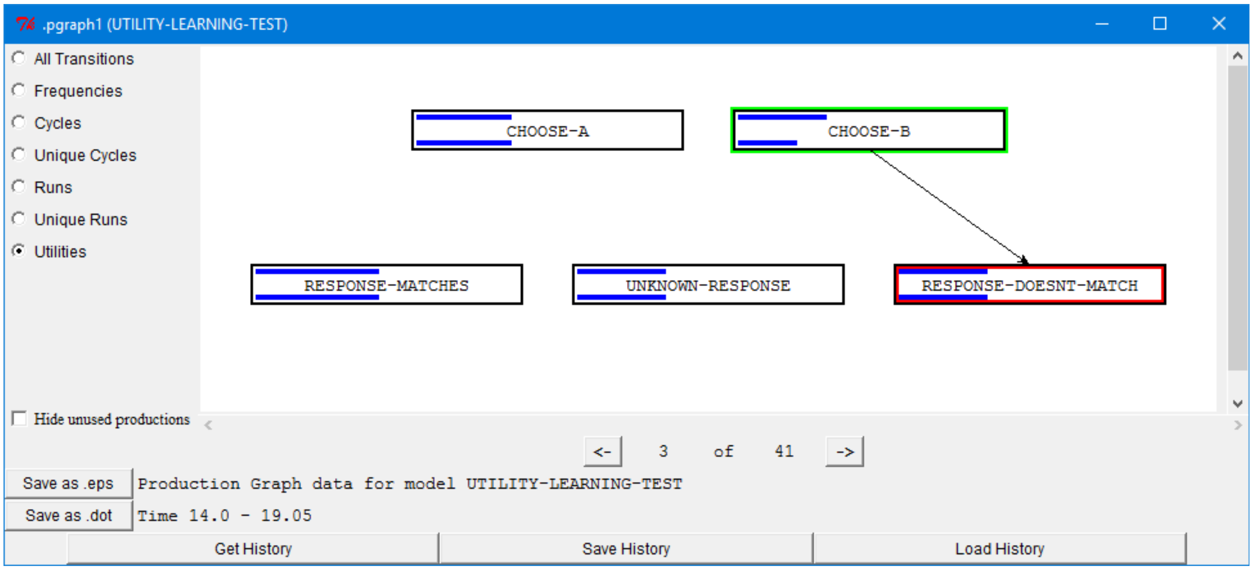
Here is the second trial where choose-a is fired and then no feedback is provided:



Production Graph data for model UTILITY-LEARNING-TEST

Time 7.0 - 12.05

On that trail we see that none of the utilities have changed. Then on the third trial we see choose-b as the first production fired followed by response-doesnt-match:



Production Graph data for model UTILITY-LEARNING-TEST

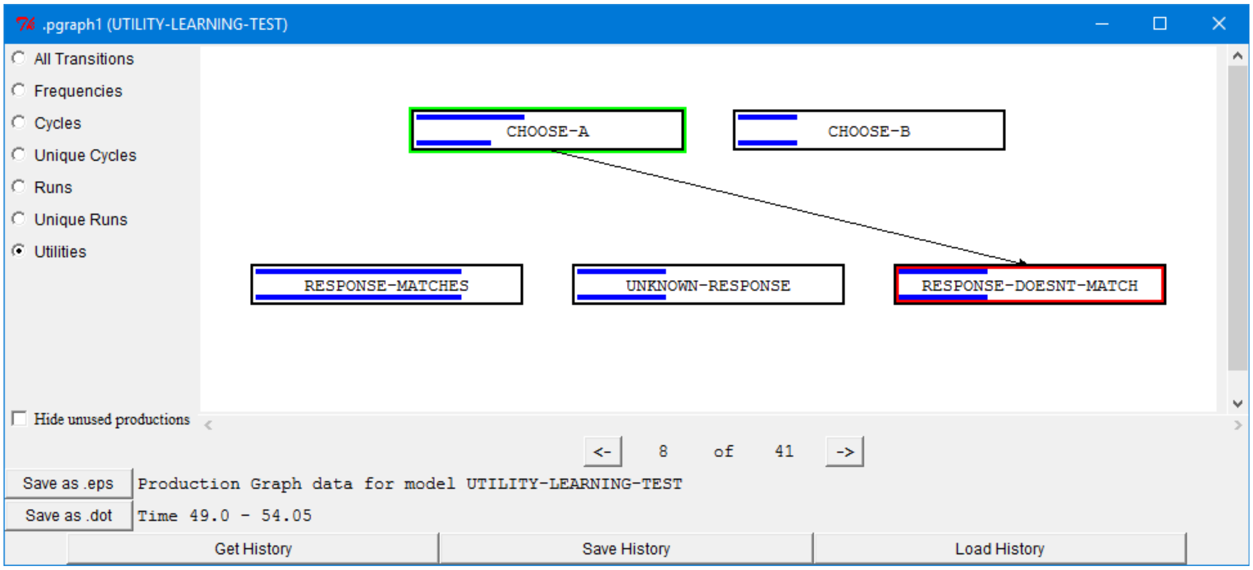
Time 14.0 - 19.05

The utility of choose-b clearly decreases, but response-doesnt-match appears to stay the same. The reason for that is because the starting utility of the productions is 0 and the reward provided by response-doesnt-match is also 0. However, it is important to note that there is still a very small change in the utility of response-doesnt-match as is shown in the utility traces displayed earlier:

```
...
    19.050   UTILITY                PROPAGATE-REWARD 0
 Utility updates with Reward = 0.0   alpha = 0.2
  Updating utility of production CHOOSE-B
   U(n-1) = 0.0   R(n) = -5.05 [0.0 - 5.05 seconds since selection]
   U(n) = -1.0100001
  Updating utility of production RESPONSE-DOESNT-MATCH
   U(n-1) = 0.0   R(n) = -0.05 [0.0 - 0.05 seconds since selection]
   U(n) = -0.010000001
```

At this time it has decreased from a utility of 0 to a utility of -0.01. That is because the effective reward for a production is the reward provided, in this case 0, minus the time since the production’s selection. Since the reward is provided when that production fires, 50ms have passed since its selection and thus the effective reward it receives is -0.05 which is multiplied by the learning rate of .2. That change in utility from 0 to -0.01 is not visible in the graphic display for this task, but might be in other tasks since the changes shown are relative to the minimum and maximum utility values in the model data.

After that trial there are several which show choose-a being selected followed by response-matches and the utilities increasing. On the eighth trial we again find choose-a being selected, but this time it is followed by response-doesnt-match:



```
Production Graph data for model UTILITY-LEARNING-TEST
Time 49.0 - 54.05
```

There we see the utility of choose-a being decreased because of the incorrect guess and again, no noticeable change in response-doesnt-match.

That is the last of the utilities graphs we will describe in the text, and ends our analysis of this test model. Before going on however you may want to look at some more of the trials in the Utilities graph and perhaps experiment with the two production tools described to get a feel for how they may be useful. When you are done, you should then call the finished function to remove the new commands which were added to implement the task:

```
? (finished)
```

```
>>> ul_issues.finished()
```

Production Compilation

Models which use production compilation will almost always be using utility learning so that the newly learned productions are introduced gradually, and they will also usually involve declarative retrievals because compiling away a retrieval is one of the major benefits in compiled productions. Because of that, one will have to be sensitive to all the issues related to those mechanisms as described above and in the unit 5 text. A recommended practice when working with production compilation is to first make sure the model works as expected without turning on production compilation. That is because it will be easier to fix the basic operation of the model as well as any procedural and declarative learning issues without having to deal with newly learned productions as well. Once the model is working well at that level, then turn on production compilation and address any new issues which arise. Those new issues may still involve general utility or activation processes in addition to issues related to the learning of new productions, but having tested the model without production compilation should make it easier to locate and address the new issues. In this text we will focus specifically on preparation, testing, and debugging issues related to the production compilation aspects of an example model, but for other modeling tasks there may be other issues which will also have to be addressed.

The Task

The task the model will perform is similar to the choice and one hit blackjack tasks from previous units. Two numbers will be presented on the screen, each from 0-3, and then one of three choices must be made using the keys s, d, and f. After a key is hit, the result of that choice for the given pair of numbers will indicate whether the result was a win, loss, or draw. The spacebar must then be pressed to advance to the next trial. No information about the choices is provided in advance and the objective is to maximize the score (wins minus losses) based on the feedback provided while responding as quickly as possible. We do not have any data for the task to fit the model to, but we do expect the model to improve both its score and response time as it plays more games. We will look at the performance of the model over the course of 200 trials, averaged into blocks of 10.

To run the model through multiple 200 trial sessions and report the average results call the pcomp-issues-game function in Lisp or the game function from the pcomp_issues module in Python. It requires one parameter which is the number of games to run and average together. It also takes an optional parameter which if specified as true will print out the results of each of the individual sessions as it runs.

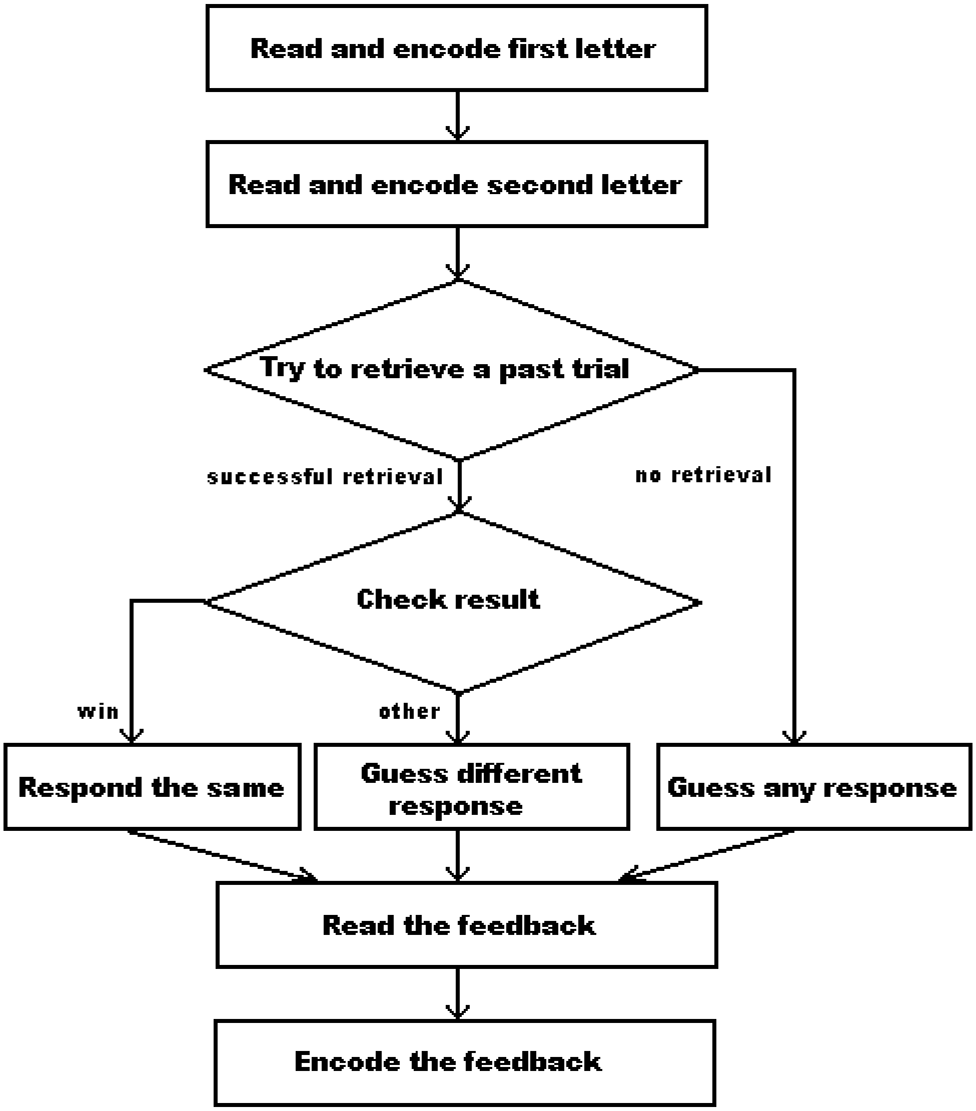
The model can also be run through fewer trials using the pcomp-issues-trials function in Lisp or the trials function from the pcomp_issues module in Python. It takes three optional parameters: the first specifies how many trials to run with the default being 200, the second indicates whether the model should be reset before the trial with the default being that it should, and the third indicates whether the scores and response times should be displayed for every 10 trials (which also defaults to being on).

The Starting Model

Before discussing anything related to production compilation, we will first describe a model which has been written to perform the task without production compilation. That model is found in the "production-compilation-issues-model.lisp" file. After that we will investigate what changes are necessary to effectively use that model with production compilation.

To model this task we have created a model which uses partial matching to retrieve a chunk stored in declarative memory from a previous trial that is similar to the current trial. This model is very similar to how the one hit blackjack model operated, and that is because this is a typical approach to use when a model must learn from experiences. For each trial of the task the model will create a chunk which includes the pair of numbers, the choice it made, and the result for that choice. Then when presented with a pair of numbers on another trial it will attempt to retrieve a chunk which indicates the winning move for the current pair and use the retrieved chunk to determine a response for this trial. The model has partial matching enabled so that it may be able to retrieve a chunk of a past trial even if this is the first time it experiences a given pair or if it has not yet found the winning move. The model also has base-level learning enabled so that the chunks which represent the trails will have their activations increased as it encounters and uses them more often which should result in a decrease in the response times over the experiment.

Here is a high-level flow chart representation of the steps which the model will be performing.



Many of those steps require multiple productions to perform, and you should be able to read through the productions in the model and follow how it works at this point. We will not describe the productions here, but we will provide some details on how the model represents the information it uses to perform the task.

Here are the chunk-types which the model specifies:

```
(chunk-type task state)
(chunk-type number visual-rep)
(chunk-type response key (is-response t))
(chunk-type trial num1 num2 result response)
```

The task chunk-type is used for the **goal** buffer to keep an explicit state marker for sequencing through the task. As has been stated in other units, doing that is not always necessary, but has been done here to make the model easier to read and follow.

The number chunk-type is needed to encode the numbers which the model will be using in its representation of the trials. That is necessary so that they can have similarities set between them. The number chunks include a slot for the visual representation so that the model can retrieve a number chunk based on the value which it gets when it attends to it on the screen. Here are the initial number chunks which the model starts with in its declarative memory:

```
(add-dm (zero isa number visual-rep "0")
        (one isa number visual-rep "1")
        (two isa number visual-rep "2")
        (three isa number visual-rep "3"))
```

The base-level of those chunks is set to a high value so that they should always be retrieved quickly. There are also similarity values set between those items using a simple linear function based on their differences.

The response chunk-type is used to represent the possible choices which the model can make in the task. It has a slot which holds the representation of the key needed to make the **manual** response and a slot with a default value of t which is there to make it easy to retrieve a random response in the model by just indicating “isa response” in the request. Here are the chunks which the model starts with in its declarative memory:

```
(add-dm (response-1 isa response key "s")
        (response-2 isa response key "d")
        (response-3 isa response key "f"))
```

Like the number chunks, those chunks are given a high base-level activation as an assumption that the model knows the instructions before starting the task.

The trial chunk-type is used to create the representation of a trial as the model performs the task. The num1 and num2 slots will contain number chunks for the trial presented. The result slot will contain one of the chunks: win, lose, or draw, and the response slot will contain the response chunk used on that trial. Here is an example of what such a chunk might look like:

```
CHUNK0-0
   RESULT  LOSE
   NUM1  ONE
   NUM2  ZERO
   RESPONSE  RESPONSE-2
```

Like the numbers, the result is encoded as a chunk so that similarities can be set between the choices. That way when the model attempts to retrieve a win, it may still be able to retrieve a draw or lose result for the trial. Since the model will not need to retrieve those result values, to keep the model simpler, they are encoded explicitly by productions instead of providing chunks in declarative memory and requiring a retrieval for encoding.

If you look at the similarity settings in the model between the result chunks you may find it curious that win is set to be more similar to lose than it is to draw. The reason for that is because if the model cannot retrieve a winning move, then retrieving a losing move is strategically better than retrieving a move which resulted in a draw for improving the score. Thus, the similarities are being used in this case to represent the usefulness of the information as an abstraction for a more deliberate strategy process in the model. That simplification is reasonable for this demonstration task since we are only concerned about showing learning through practice, but a more thorough model of a real task like this may require the model to account for that strategy processing.

Here are the parameter settings from the model:

```
  (sgp :esc t :lf .5 :bll .5 :mp 18 :rt -3 :ans .25)
```

Since we do not have data to fit, the parameters for the model were either set to recommended values (:bll and :ans) or simply adjusted to values which resulted in showing improvements which seemed reasonable for the demonstration.

Here are the results for the model on the task averaged over 50 runs:

```
Average Score of 50 trials
2.44 4.96 6.34 6.76 7.28 7.40 7.72 8.04 7.82 8.02 8.00 8.42 8.30 8.76 8.18 8.44 8.32 8.54 8.80 8.44
Average Response times
7.87 4.79 3.13 2.41 2.12 1.86 1.73 1.59 1.52 1.48 1.42 1.34 1.30 1.24 1.25 1.23 1.24 1.19 1.18 1.16
```

It is improving its score and getting faster over trials, which is what we expect. You may want to step through the model and perhaps explore its operation with the history and graphing tools before continuing so that you have a good understanding of how it operates.

Considerations for Production Compilation

When using production compilation, there are some things that should be considered with respect to the model for production compilation to work well. If the model was written with those considerations in mind, then the next step would be to turn production compilation on and start testing. However, if the model was not written for use with production compilation, which is the case for the starting model we described above, then just turning it on "to see what happens" is usually not going to work very well. For example, here is what happens if we run the starting model with production compilation enabled and no other changes:

```
Average Score of 50 trials
1.50 1.90 2.40 2.58 3.02 2.66 3.54 3.10 3.78 4.00 3.76 3.56 3.58 3.40 4.28 4.04 4.26 4.08 4.12 3.98
Average Response times
8.57 6.37 5.45 5.10 4.45 4.53 3.74 3.80 3.59 3.45 3.38 3.20 3.32 3.24 3.03 2.92 2.74 2.85 2.75 2.71
```

Neither the scores nor the response times look very good relative to how it ran previously. That might suggest that one should then start tracing, testing, and debugging the model, but the recommendation would be to first consider the following issues before attempting to run it with production compilation.

What is the task and how is the model run

Production compilation requires repetition to be effective because it will only show a change if the model has the opportunity to use the newly learned productions. Thus, the task must be one in which the model will be running repeatedly without being reset. Models which are already using base-level learning or utility learning will likely have that characteristic already. Other tasks however may not, for example the fan effect model and the subitizing models from the tutorial do not. Those models are being reset for each trial, and thus production compilation would not show any change in the results which they produce. If one wanted to use models like that with production compilation they would have to be changed so that they run continuously over the trials instead. The other thing to consider is whether there is enough repetition in the task to be effective. If one is looking for declarative knowledge to become encapsulated in the productions there will typically need to be multiple usages of those chunks so that the productions can be strengthened to the point of competing with the originals. For example, even if the fan experiment model were to be changed to present the trials continuously, since each test sentence is only presented once to that model, there would probably not be any use of the productions which proceduralize the declarative information. If the task is not continuous and/or does not provide any repetition then there is little reason to enable production compilation since it will not affect the operation of the model.

Utility learning

One of the most important issues with respect to production compilation is utility learning. It is the learning of utilities for the new productions which leads to their gradual introduction and whether they will end up being used in place of the original productions. Without utility learning the new productions will only ever have their initial utility value. If the model has not set the initial utilities for the existing productions or changed the :nu and :iu parameters then a newly learned production will have the same utility, 0, as all other productions and immediately compete with them, regardless of whether that production is actually useful or not. For example, in this task, that might mean that a production which always makes a losing move may be competing equally with the productions which attempt to remember a past move.

If the starting model was already using utility learning then one will want to make sure that the newly learned productions will start out with lower utilities than the original ones. If the original productions have greater than zero utilities (either because they are explicitly set or because the :iu parameter was set to greater than zero) then no immediate change would be needed. If the original productions do start with zero or negative utilities then the :nu parameter, which controls where the newly learned productions' utilities start, should be set to a negative value so that they are lower than the original productions' utilities. In either case those initial utility values may need to be adjusted as one starts to test the model, but it helps to have a reasonable starting point.

If the original model did not use utility learning, which is true for the starting model we have here, then one will first have to add that to it. That means that in addition to enabling the mechanism one will have to add some rewards to the model so that it has opportunities for learning. The utility values for the initial productions and starting values for the newly learned productions will also have to be set so that the new productions start below the originals (as described above).

When enabling utility learning for a model which will be using production compilation one will also want to make sure that there is some utility noise in the system so that the newly learned productions will have a chance to be selected. If there is no noise then the new productions will never exceed the utility of their parents (assuming a recommended utility learning rate of less than 1.0) and thus will never be selected. The amount of utility noise will affect the rate at which the new productions get used (how many times they will need to be recreated before they have utilities with a reasonable probability of being selected) since the noise affects the probability of selecting the productions as shown in the equation from unit 6. Assuming that one wants the productions to be introduced gradually, a low value for the noise is recommended, but what exactly constitutes a “low” value will depend on the relative utilities and the learning rate in the model.

Expected Changes

Another thing to consider for a model is what production compilation may change about the way that it operates. There are two very general things that production compilation can do: reduce sequences of production firings into fewer productions and transition knowledge from a declarative representation into a procedural one. Those can combine to produce interesting results, like the over generalization that occurs in the past-tense model, but particular effects like that usually require careful planning in the design of the model. As a first step, particularly for a model which may not have been specifically designed for production compilation, just considering the potential changes production compilation may have can be helpful before trying to use it.

If the model is being designed from the start to utilize production compilation, then knowing what effects are desired will help to shape the initial creation of the model. When looking to get a decrease in the time the model takes because of a reduction of long production sequences one will likely want to start the model with productions which perform small steps so that there are opportunities for productions to be compiled together. One will also want to be careful about separating perceptual and motor tasks which will block the compilation of productions from those which are expected to be compiled together. If the proceduralization of declarative knowledge is desired, obviously one will first have to have a model which makes requests for declarative information. Then one will have to carefully consider the productions which request and harvest the **retrieval** buffer chunks. Those productions will need to be safe for compilation, and thus will need to avoid other actions like requesting and harvesting perceptual information or performing multiple motor actions since those cannot be combined through production compilation. In addition to that, one may want to consider the details of what information is used to make the requests and what is tested in the harvesting productions. Those details will shape how the compiled production works and are important when looking for particular results, like generalization.

If the model was not initially designed for production compilation, then one should look over the model with respect to the issues noted above to determine if compilation is going to be effective at performing the desired results. If a speed up from creating shorter production sequences is desired, then one will want to look at the productions and see if they seem amenable to compilation. Things to look for are whether the productions are already performing multiple actions which might prevent them being combined any further and whether or not the perceptual and motor actions are isolated or pervasive throughout the productions. If it does not look like there will be many opportunities for compilation to combine productions further then one may want to consider making some changes to provide those opportunities. That might involve breaking up existing productions to make the model slower initially so that production compilation can provide the speed up. It may also require creating productions specifically for the perceptual and motor actions so that they are separated from productions which can be compiled together. If the transition from declarative to procedural knowledge is desired, then, like above, one will want to look at the productions which request and harvest the declarative chunks to make sure that they can be composed.

Considering the starting model

With those concepts in mind, we will look at the task and starting model before enabling production compilation and running it again. Because the original task involved base-level learning the model already ran continuously over the trials. Also, the 200 trials provided enough repetition to show learning for the declarative information. So the task and model seem like they are functionally capable of working with production compilation.

Let us next consider what we expect production compilation to do for this model. Looking over the productions, this starting model has been written with productions which already combine multiple actions. In addition, there are only 10 productions fired to perform a trial of the task as it stands, and since many of those productions are involved with perceptual processes that will always be required there appears to be little opportunity for this model to improve performance from reducing long production sequences. If we were interested in fitting a particular gradual performance increase, then we may want to reconsider this as a starting model and perhaps simplify those productions or move to a model which uses a more general instruction following process to do the task, like the paired associate task from unit 7. For this example we will not make any changes to try to change that and just see if there are any gains in that respect as is. Transitioning the knowledge from declarative to procedural however does seem like something which would be desirable in this task. Instead of always having to retrieve a move from declarative memory we would like to see this model develop productions which are able to make a move directly. The productions which the model has for performing the critical retrievals are free of perceptual and motor actions (other than a final response). Therefore, it seems like it should be possible for this model to do that. We could look more closely at those productions now to make sure that they can safely be composed, but instead we will wait and let the production compilation mechanism itself indicate any problems it finds when we run it.

The last thing to consider is utility learning, and this starting model does not currently use it. Therefore we will need to add that to it before production compilation will be able to affect the operation of the model through a gradual introduction of newly learned productions. That will involve setting some general parameters as well as providing rewards to the model. Because the model’s results did not depend on utility learning we will have to start by just setting some reasonable values, and then perhaps adjust them later once we enable production compilation and see how it performs. We will take a little time to walk through exactly how we will chose those initial values in the next few paragraphs.

Since the model already has three productions for processing the feedback, that seems like a good place to add rewards. To determine how much reward to provide, we will make some simple assumptions and go from there. If we assume that new productions will start at a utility of 0 (the default), we will want the initial productions to start somewhere above that. Another assumption that is usually a good one to make is that we do not want the initial productions to drop to a utility below where a newly created production starts since we do not want the newly learned productions to immediately be preferred. Since we are assuming that new productions start with a utility of 0, that means that the initial productions should always have positive utilities. To ensure that, we do not want productions to get negative effective rewards (the reward minus the time between the production selection and the reward being provided). Thus, the minimum reward we want to provide to the model will depend on the longest time we expect the model to take before getting a reward. That should happen on the first trial it does because that will result in a retrieval failure for a past game, which represents the maximum time a retrieval can take. To find that we will turn on the trace with the detail level set to low to see when the feedback production fires and run one trail (since the :seed parameter is not set in the starting model when you run it your trace will differ slightly from the one shown here):

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TASK0 NIL
     0.000   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED DETECT-TRIAL-START
     0.135   VISION                 SET-BUFFER-CHUNK VISUAL TEXT0
     0.185   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-1
     0.188   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.250   IMAGINAL               SET-BUFFER-CHUNK IMAGINAL CHUNK0
     0.300   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-1
     0.300   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1
     0.350   PROCEDURAL             PRODUCTION-FIRED FIND-NUM-2
     0.435   VISION                 SET-BUFFER-CHUNK VISUAL TEXT1
     0.485   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-2
     0.489   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL TWO
     0.539   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-2
     0.589   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-PAST-TRIAL
    10.632   DECLARATIVE            RETRIEVAL-FAILURE
    10.682   PROCEDURAL             PRODUCTION-FIRED NO-PAST-TRIAL
    10.684   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL RESPONSE-2
    10.734   PROCEDURAL             PRODUCTION-FIRED RESPOND
    10.734   MOTOR                  PRESS-KEY KEY d
    10.944   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2 NIL
    11.079   PROCEDURAL             PRODUCTION-FIRED DETECT-FEEDBACK
    11.164   VISION                 SET-BUFFER-CHUNK VISUAL TEXT2
    11.214   PROCEDURAL             PRODUCTION-FIRED ENCODE-FEEDBACK-LOSE
    11.214   MOTOR                  PRESS-KEY KEY SPACE
    11.214   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK1
    11.414   ------                 Stopped because no events left to process
```

It takes the model a little under 11 seconds to respond, and the feedback production fires at time 11.214 seconds. Therefore if we want all of the productions to receive positive rewards on all of the trials we will need to provide a reward greater than 11.214 in each case. Then, as long as the model responds, all of the productions will receive a positive reward and should not drop to a utility below 0.

We now need to decide exactly how much reward to provide for each result, and we will also need to consider the starting utility of the initial productions. What values to use can depend on many factors in a complex model, but in this case we will use the minimum reward value for a positive reward found above to provide some guidance. Thinking about the expected result, learning productions which respond without retrieving a past game, presumably we only really want to learn such productions for the responses which lead to wins, and not losses or draws. To achieve that we will want to have multiple reward values so that wins are favored over the others. Whether or not to favor a draw over a loss might matter for fitting real performance, but for this task we will assume that a draw is better than a loss. Thus, we will have three reward values provided to the model. Since we want all of the productions to receive positive rewards for completing the task, we will start by giving a loss a reward of 12. From there we will choose some larger values for a draw and a win. One could perform some analysis to determine values based on probability of being selected as a function of rewards, but since we do not exactly know how production compilation will affect this specific model we will just choose values of 15 and 18 for a draw and win respectively so that there is some distance between them and see how that works. Thus, here are the settings which we will add to the model:

```
  (spp encode-feedback-win :reward 18)
  (spp encode-feedback-lose :reward 12)
  (spp encode-feedback-draw :reward 15)
```

Now we need to choose the starting utility for the initial productions. Given the nature of the task and the rewards chosen already, starting with the initial productions having a utility equal to the reward given for a draw seems like a good place to start them. Then a win should result in increasing utilities while a loss will cause them to decrease.

The last thing we need to add is the noise. As with the rewards, we could try to determine a value analytically, but instead we will just pick a starting noise value of 1.0 and adjust it later if we notice any issues. We will leave the learning rate, alpha, at its default of .2. So, here are the settings which we need to add to the model now to enable utility learning and set those parameters:

```
(sgp :ul t :egs 1.0 :iu 15)
```

Those changes should not affect the operation of the current model since it does not have any productions which are currently competing for selection based on utility. If we run a few trials to check it still seems to be performing as before:

```
Average Score of 10 trials
1.70 6.20 7.80 7.90 7.50 8.50 8.30 9.10 9.50 8.60 9.40 9.10 8.70 8.80 9.30 9.30 8.90 9.40 8.90 9.40
Average Response times
8.78 5.02 3.38 2.40 2.06 1.88 1.51 1.50 1.30 1.33 1.28 1.25 1.27 1.24 1.21 1.23 1.18 1.20 1.18 1.15
```

You may want to inspect that in more detail using the Environment tools as described for the first model above to verify that it is always receiving a reward and to see how the utilities are changing, even though they are not affecting the operation of this model.

Now that we have inspected the model and made the changes that were necessary for production compilation to work well it is time to enable production compilation and start testing. To enable production compilation all we need to do is set the parameter :epl to t, but we are also going to turn on the additional trace output it provides so that we can see what it does as the model runs. So we will add this additional setting to the model:

```
(sgp :epl t :pct t)
```

Testing the Model

Testing a model which uses production compilation typically involves four phases. The first is making sure the model performs as expected without production compilation being turned on. After that, production compilation is turned on and one runs the model watching the productions which are generated by production compilation. The objective here is to verify that things are working well at the symbolic level. You want to make sure that production compilation is able to compose the starting productions into new productions, and that those new productions appear to be doing the things you expect. Once it looks like production compilation is producing reasonable new productions you want to make sure that those new productions are not going to cause problems for the model's operation. If the model is small and does not require a long time to run, then it may be sufficient to just run it for multiple trials and monitor its operation, but for a large or very long running model it may be easier to temporarily adjust some of the model parameters so that the new productions are used right away so that their effects are easier to see. Finally, once you are comfortable with the productions generated through compilation and how they affect the model's basic operation you can then start to run the model for comparison to data and determining whether or not you get the overall results you were looking for and attempt to adjust the parameters as needed to fit your data. As with all testing and debugging, that is not always going to be a simple sequential process since one may have to go back and perform earlier tests again because of changes or problems which are encountered in a later step.

Since we have already tested the model without production compilation we will now turn on compilation and look at the productions it generates and the places where it cannot generate productions. To see that we will need to turn the model trace on, the :v parameter, in addition to the production compilation trace value we just added. Since we may also want to be able to repeat the same trials again it is a good idea to have the model print out the current seed when it gets reset so we can set that value again later if we want to run a trial again to analyze. To do that one would add this setting to the top of the model definition:

```
(sgp :seed)
```

However, so that your model runs correspond to those shown in the text we will be setting explicit seed values in the model for testing purposes. The first seed we will use is this one:

```
(sgp :seed (1 3))
```

Now we will run the model one trial at a time to look at the results of production compilation. It will require multiple trials before we are going to see the primary result we are expecting because the model will have to first learn a chunk which represents a trial and then be able to successfully retrieve it. We could adjust the parameters and add additional chunks to the model to artificially create the situations we are interested in seeing production compilation applied to, but since this is a fairly simple model that is not really necessary because we can easily investigate the situations occurring under the model's normal operation.

Here is what we get with the production compilation trace enabled for the first trial with the :trace-detail set to low:

```
     0.000   GOAL                   SET-BUFFER-CHUNK GOAL TASK0 NIL
     0.000   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0 NIL
     0.050   PROCEDURAL             PRODUCTION-FIRED DETECT-TRIAL-START
Production Compilation process started for DETECT-TRIAL-START
  No previous production to compose with.
  Setting previous production to DETECT-TRIAL-START.
     0.135   VISION                 SET-BUFFER-CHUNK VISUAL TEXT0
     0.185   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-1
Production Compilation process started for ATTEND-NUM-1
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-TRIAL-START and ATTEND-NUM-1 cannot be composed.
  Setting previous production to ATTEND-NUM-1.
     0.187   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL THREE
     0.250   IMAGINAL               SET-BUFFER-CHUNK IMAGINAL CHUNK0
     0.300   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-1
Production Compilation process started for ENCODE-NUM-1
  Production ATTEND-NUM-1 and ENCODE-NUM-1 are being composed.
  New production:

(P PRODUCTION0
  "ATTEND-NUM-1 & ENCODE-NUM-1 - THREE"
   =GOAL>
       STATE ATTEND-NUM-1
   =IMAGINAL>
   =VISUAL>
       VALUE "3"
 ==>
   =IMAGINAL>
       NUM1 THREE-0
   =GOAL>
       STATE FIND-NUM-2
   +VISUAL-LOCATION>
       :ATTENDED NIL
    >  SCREEN-X CURRENT
)
Parameters for production PRODUCTION0:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-1.
     0.300   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1
     0.350   PROCEDURAL             PRODUCTION-FIRED FIND-NUM-2
Production Compilation process started for FIND-NUM-2
  Buffer VISUAL-LOCATION prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production ENCODE-NUM-1 and FIND-NUM-2 cannot be composed.
  Setting previous production to FIND-NUM-2.
     0.435   VISION                 SET-BUFFER-CHUNK VISUAL TEXT1
     0.485   PROCEDURAL             PRODUCTION-FIRED ATTEND-NUM-2
Production Compilation process started for ATTEND-NUM-2
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production FIND-NUM-2 and ATTEND-NUM-2 cannot be composed.
  Setting previous production to ATTEND-NUM-2.
     0.487   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL ZERO
     0.537   PROCEDURAL             PRODUCTION-FIRED ENCODE-NUM-2
Production Compilation process started for ENCODE-NUM-2
  Production ATTEND-NUM-2 and ENCODE-NUM-2 are being composed.
  New production:

(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM2 ZERO-0
   =GOAL>
```

```
       STATE RETRIEVE-PAST-TRIAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-2.
     0.587   PROCEDURAL             PRODUCTION-FIRED RETRIEVE-PAST-TRIAL
Production Compilation process started for RETRIEVE-PAST-TRIAL
  Production ENCODE-NUM-2 and RETRIEVE-PAST-TRIAL are being composed.
  New production:

(P PRODUCTION2
  "ENCODE-NUM-2 & RETRIEVE-PAST-TRIAL"
   =GOAL>
       STATE ENCODE-NUM-2
   =IMAGINAL>
       NUM1 =N1
   =RETRIEVAL>
 ==>
   =IMAGINAL>
       NUM2 =RETRIEVAL
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   +RETRIEVAL>
       NUM1 =N1
       NUM2 =RETRIEVAL
       RESULT WIN
)
Parameters for production PRODUCTION2:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-PAST-TRIAL.
    10.630   DECLARATIVE            RETRIEVAL-FAILURE
    10.680   PROCEDURAL             PRODUCTION-FIRED NO-PAST-TRIAL
Production Compilation process started for NO-PAST-TRIAL
  Cannot compile RETRIEVE-PAST-TRIAL and NO-PAST-TRIAL because the time between them exceeds the
threshold time.
  Setting previous production to NO-PAST-TRIAL.
    10.682   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL RESPONSE-2
    10.732   PROCEDURAL             PRODUCTION-FIRED RESPOND
Production Compilation process started for RESPOND
  Production NO-PAST-TRIAL and RESPOND are being composed.
  New production:

(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
   =GOAL>
```

```
        STATE DETECT-FEEDBACK
    +MANUAL>
        CMD PRESS-KEY
        KEY "d"
)
Parameters for production PRODUCTION3:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
    10.732   MOTOR                  PRESS-KEY KEY d
    10.942   VISION                 SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2 NIL
    11.077   PROCEDURAL             PRODUCTION-FIRED DETECT-FEEDBACK
Production Compilation process started for DETECT-FEEDBACK
  Production RESPOND and DETECT-FEEDBACK are being composed.
  New production:

(P PRODUCTION4
  "RESPOND & DETECT-FEEDBACK"
   =GOAL>
       STATE RESPOND
   =IMAGINAL>
   =RETRIEVAL>
       IS-RESPONSE T
       KEY =KEY
   =VISUAL-LOCATION>
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE =RETRIEVAL
   =GOAL>
       STATE ENCODE-FEEDBACK
   +VISUAL>
       CMD MOVE-ATTENTION
       SCREEN-POS =VISUAL-LOCATION
   +MANUAL>
       CMD PRESS-KEY
       KEY =KEY
)
Parameters for production PRODUCTION4:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to DETECT-FEEDBACK.
    11.162   VISION                 SET-BUFFER-CHUNK VISUAL TEXT2
    11.212   PROCEDURAL             PRODUCTION-FIRED ENCODE-FEEDBACK-LOSE
Production Compilation process started for ENCODE-FEEDBACK-LOSE
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-FEEDBACK and ENCODE-FEEDBACK-LOSE cannot be composed.
  Setting previous production to ENCODE-FEEDBACK-LOSE.
    11.212   MOTOR                  PRESS-KEY KEY SPACE
    11.212   GOAL                   SET-BUFFER-CHUNK GOAL CHUNK1
    11.412   ------                 Stopped because no events left to process
```

After every production fires production compilation attempts to create a new production, and for each attempt the production compilation trace provides the details of what the resulting new production looks like or a description of an issue which prevented it from compiling the productions. We will look at each one that occurred in this trace to make sure things are working as expected.

Here is the first production compilation trace message:

```
Production Compilation process started for DETECT-TRIAL-START
  No previous production to compose with.
  Setting previous production to DETECT-TRIAL-START.
```

Since that is the first production there is nothing to compose it with and thus all it can do is record that that production is now the previous one for use when the next one fires. The next result is this:

```
Production Compilation process started for ATTEND-NUM-1
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-TRIAL-START and ATTEND-NUM-1 cannot be composed.
  Setting previous production to ATTEND-NUM-1.
```

It indicates that the productions cannot be composed because the visual buffer blocks it due to the request and harvesting of a chunk. Since perceptual information cannot be compiled into new productions that is what we would expect and there is not anything we need to do to try to fix that.

The next result is somewhat unexpected:

```
Production Compilation process started for ENCODE-NUM-1
  Production ATTEND-NUM-1 and ENCODE-NUM-1 are being composed.
  New production:

(P PRODUCTION0
  "ATTEND-NUM-1 & ENCODE-NUM-1 - THREE"
   =GOAL>
       STATE ATTEND-NUM-1
   =IMAGINAL>
   =VISUAL>
       VALUE "3"
 ==>
   =IMAGINAL>
       NUM1 THREE-0
   =GOAL>
       STATE FIND-NUM-2
   +VISUAL-LOCATION>
       :ATTENDED NIL
    >  SCREEN-X CURRENT
)
Parameters for production PRODUCTION0:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-1.
```

Since the encoding step which the model performs requires retrieving the number chunk from declarative memory, production compilation is able to compose those two into a new production which does not require the retrieval. We did not really consider that in what we expected from the model, but it appears to be another opportunity for the model to get faster over time which is in line with what we want so having such a production does not seem to be a problem.

The next two production compilation attempts are unsuccessful because the productions involved are performing perceptual actions:

```
Production Compilation process started for FIND-NUM-2
  Buffer VISUAL-LOCATION prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production ENCODE-NUM-1 and FIND-NUM-2 cannot be composed.
  Setting previous production to FIND-NUM-2.

Production Compilation process started for ATTEND-NUM-2
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production FIND-NUM-2 and ATTEND-NUM-2 cannot be composed.
  Setting previous production to ATTEND-NUM-2.
```

After that is a production very similar to production0 this time encoding the second number into the imaginal chunk without having to perform the retrieval:

```
Production Compilation process started for ENCODE-NUM-2
  Production ATTEND-NUM-2 and ENCODE-NUM-2 are being composed.
  New production:

(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM2 ZERO-0
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
)
Parameters for production PRODUCTION1:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-2.
```

Again, this was not expected, but seems to be in line with the general expectations.

The next composition results in a production which is just the composition of two productions without removing an intervening retrieval:

```
Production Compilation process started for RETRIEVE-PAST-TRIAL
  Production ENCODE-NUM-2 and RETRIEVE-PAST-TRIAL are being composed.
  New production:

(P PRODUCTION2
  "ENCODE-NUM-2 & RETRIEVE-PAST-TRIAL"
   =GOAL>
       STATE ENCODE-NUM-2
   =IMAGINAL>
       NUM1 =N1
   =RETRIEVAL>
 ==>
   =IMAGINAL>
       NUM2 =RETRIEVAL
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   +RETRIEVAL>
       NUM1 =N1
       NUM2 =RETRIEVAL
       RESULT WIN
)
Parameters for production PRODUCTION2:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-PAST-TRIAL.
```

This is another opportunity for the model to speed up over time, and also in line with the general expectation for the model.

Next, we see a failure to compose productions because of the amount of time that passed:

```
Production Compilation process started for NO-PAST-TRIAL
   Cannot compile RETRIEVE-PAST-TRIAL and NO-PAST-TRIAL because the time between them exceeds the threshold time.
  Setting previous production to NO-PAST-TRIAL.
```

The threshold time is a settable parameter in the model which we might what to consider adjusting, but since there was also a failure to retrieve a chunk those productions would not have been composable anyway. So, we will hold off on adjusting the parameter until we see whether or not successful retrievals are taking too long.

The next opportunity for composition results in a production which eliminates another retrieval:

```
Production Compilation process started for RESPOND
  Production NO-PAST-TRIAL and RESPOND are being composed.
  New production:

(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
Parameters for production PRODUCTION3:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
```

This production effectively results in guessing “d” when it cannot remember a past move. While that does save time by eliminating a production and a retrieval, it probably will not be a very useful production overall and we may never see it actually being used.

Despite the number of different conditions involved across various cognitive, perceptual, and motor modules the respond and detect-feedback productions are able to be composed:

```
Production Compilation process started for DETECT-FEEDBACK
  Production RESPOND and DETECT-FEEDBACK are being composed.
  New production:

(P PRODUCTION4
  "RESPOND & DETECT-FEEDBACK"
   =GOAL>
       STATE RESPOND
   =IMAGINAL>
   =RETRIEVAL>
       IS-RESPONSE T
       KEY =KEY
   =VISUAL-LOCATION>
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE =RETRIEVAL
   =GOAL>
       STATE ENCODE-FEEDBACK
   +VISUAL>
       CMD MOVE-ATTENTION
       SCREEN-POS =VISUAL-LOCATION
   +MANUAL>
       CMD PRESS-KEY
       KEY =KEY
)
Parameters for production PRODUCTION4:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to DETECT-FEEDBACK.
```

This seems like it might be yet another helpful production to save time doing the task, but a careful look at the conditions and actions with respect to what happens in the task will expose an issue with this production. This production gets a visual-location buffer test from detect-feedback and the manual action from respond. However, a chunk only enters the **visual-location** buffer because of buffer stuffing after the model makes a response which causes the feedback to appear. Thus, while there is nothing syntactically wrong with production4 it will never be able to match during this task since it has conditions which only result from actions it performs. That happens because production compilation has no way to detect dependencies which occur outside of the productions, in this case that the screen changes as a result of the key press, and thus it creates a production which will never be able to fire. Typically, that will not be problematic since a production which does not match has no effect on the model's performance, but in some rare situations it may be necessary to explicitly indicate dependencies of that nature somehow in the production conditions to avoid the composition of productions which violate implicit task dependencies.

Here is the final opportunity for composition in this trial:

```
Production Compilation process started for ENCODE-FEEDBACK-LOSE
  Buffer VISUAL prevents composition of these productions
   because the first production makes a request and the second production harvests the chunk.
  Production DETECT-FEEDBACK and ENCODE-FEEDBACK-LOSE cannot be composed.
  Setting previous production to ENCODE-FEEDBACK-LOSE.
```

which fails because of the perceptual action involved.

Looking at the first trial produced a couple of unexpected compositions, but nothing which seems to violate what we want the model to do overall. Now we will run a couple more trials looking for compositions we have not seen yet, and in particular we want to see what happens when there is a successful retrieval of a past trial. We need to make sure to run those additional trials without resetting the model, thus we will need to specify the optional reset value as nil/False so as to not reset the model:

```
? (pcomp-issues-trials 1 nil)
```

```
>>> pcomp_issues.trials(1,False)
```

The second trail still does not result in a successful retrieval, but there are a few new production compilation attempts worth looking at. The first occurs immediately when the feedback encoding production of the previous trial gets composed with the detect-trial-start production:

```
Production Compilation process started for DETECT-TRIAL-START
  Production ENCODE-FEEDBACK-LOSE and DETECT-TRIAL-START are being composed.
  New production:

(P PRODUCTION5
  "ENCODE-FEEDBACK-LOSE & DETECT-TRIAL-START"
   =GOAL>
       STATE ENCODE-FEEDBACK
   =IMAGINAL>
   =VISUAL-LOCATION>
   =VISUAL>
       VALUE "lose"
   ?IMAGINAL>
       STATE FREE
   ?MANUAL>
       STATE FREE
   ?VISUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESULT LOSE
   +VISUAL>
       CMD MOVE-ATTENTION
       SCREEN-POS =VISUAL-LOCATION
   +MANUAL>
```

```
       CMD PRESS-KEY
       KEY SPACE
   +IMAGINAL>
   +GOAL>
       STATE ATTEND-NUM-1
)
Parameters for production PRODUCTION5:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward 15.000
 :fixed-utility    NIL
  Setting previous production to DETECT-TRIAL-START.
```

Like production4 from the first trial this production has a visual-location condition which will not be satisfied while doing this task because it comes about from the action which this production would make. Thus, this is another production which will never match and fire.

A couple of other new productions are also composed and they appear similar to those created on the first trial to compose the declarative information in the encoding phase into productions, which again seems reasonable.

Later in the run we see two occasions where production compilation recreates the same productions which it did in the first trial:

```
Production Compilation process started for RETRIEVE-PAST-TRIAL
  Production ENCODE-NUM-2 and RETRIEVE-PAST-TRIAL are being composed.
  Recreating production PRODUCTION2
Parameters for production PRODUCTION2:
 :utility -1.959
 :u   2.451
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVE-PAST-TRIAL.

Production Compilation process started for DETECT-FEEDBACK
  Production RESPOND and DETECT-FEEDBACK are being composed.
  Recreating production PRODUCTION4
Parameters for production PRODUCTION4:
 :utility    NIL
 :u   2.859
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to DETECT-FEEDBACK.
```

In both those cases we see that the true utility (:u value) of those productions has now increased from 0, their initial value when first composed, since they get rewards based on the parent productions' utilities with each recreation.

There is however one curious composition given what we saw with the first trial:

```
(P PRODUCTION9
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
```

```
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-1
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
Parameters for production PRODUCTION9:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
```

On the first trial we also saw the composition of a production which collapsed no-past-trial with respond removing the retrieval of the chunk response-2:

```
(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
```

So the question is why on this trial is production9 created instead of just strengthening production3? If we look closely at those productions we can see that they differ slightly in the modifications that they perform to the chunk in the **imaginal** buffer:

```
   =IMAGINAL>
       RESPONSE RESPONSE-2-1
```

and

```
   =IMAGINAL>
       RESPONSE RESPONSE-2-0
```

So, now the question is why do they differ like that? If we look at declarative memory we do not find either of those chunks. That probably means that they have been merged with other chunks. We can find that out using the pprint-chunks/pprint_chunks command to display them:

```
RESPONSE-2-0 (RESPONSE-2)
   KEY  "d"
   IS-RESPONSE  T
```

```
RESPONSE-2-1 (RESPONSE-2)
   KEY  "d"
   IS-RESPONSE  T
```

Both have been merged with the original chunk response-2, which does not seem to help explain why those are different productions. To answer that, we will have to look at where that action comes from in the original productions.

The modification to the imaginal chunk is an action from the respond production:

```
  (p respond
     =goal>
       isa task
       state respond
     =retrieval>
       isa response
       key =key
     ?manual>
       state free
     =imaginal>
       isa trial
     ==>
     =imaginal>
       response =retrieval
     +manual>
       cmd press-key
       key =key
     =goal>
       state detect-feedback)
```

In that production the slot is set to the chunk which is currently in the **retrieval** buffer. Recall that buffers hold copies of chunks. Thus, when the respond production fires the chunk in the **retrieval** buffer is not the chunk response-2 but a copy of it and since the buffer has not yet been cleared (that happens after respond fires) that copy in the buffer has not yet been merged with response-3. Production compilation does not know anything about what will happen to chunks in the future when it uses them in composing a production. Therefore, every time production compilation combines those two productions the chunk in the **retrieval** buffer will always be a new chunk and since that chunk is used to set the response slot of the **imaginal** buffer it must create a new production each time.

That may seem like a flaw with production compilation, but since it is not plausible for the mechanism to know the future that is all it can do. Therefore the flaw is really in the model design – specifically the representation of the knowledge it is using. It is the content of chunks which should be meaningful to the model, not their particular identity. While it is often convenient to refer to chunks by name like that in a model, there are situations where such shortcuts are inappropriate and should be avoided. There are a lot of ways that this model could be changed to not use the identity of the retrieved chunk directly, but since having those separate productions from this composition should not affect what we expect from the model we are not going to make any of those changes right now. However, if we encounter any other similar issues we will reconsider changing the model.

Since we still have not seen a successful retrieval we will run the model for a few more trials until we get one. On the fifth trial the model successfully retrieves a past trial chunk:

```
    48.372   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL CHUNK6-0
    48.422   PROCEDURAL             PRODUCTION-FIRED RETRIEVED-A-WIN
Production Compilation process started for RETRIEVED-A-WIN
  Production RETRIEVED-A-WIN is not valid for compilation
   because it has an indirect action with the RETRIEVAL buffer
```

Unfortunately, production compilation tells us that the retrieved-a-win production is invalid for compilation purposes because it makes an indirect retrieval action. Here is that production:

```
(p retrieved-a-win
   =goal>
     isa task
     state process-past-trial
   =retrieval>
     isa trial
     result win
     response =response
   ==>
   +retrieval> =response
   =goal>
     state respond)
```

The value from the response slot of the trial chunk is being used to specify the retrieval, which is done as a consequence of a specific chunk reference being stored in that slot as was discussed for the respond production above. Since composing retrieve-past-trial and retrieved-a-win is something that we want the model to do we are going to have to change the representation stored in the response slot of the trial chunks and the productions which use them.

There are many ways which we could change the model, but because the model has such a simple representation for the response chunks we will start by making a small change and see how that affects things. The change that we will make is that instead of storing a response chunk itself in the response slot of the trial chunk we will store the value from the key slot of a response chunk in the response slot of the trial chunk. If the response chunks had contained more slots, then this simple change may not have been possible and a more thorough analysis of the model and its representations would have been required to determine how to request and harvest the chunks needed so that they would be compatible with production compilation.

Making that change requires changing three productions. The respond production needs to be changed to save the key slot's value instead of the response chunk itself:

```
(p respond
   =goal>
     isa task
     state respond
   =retrieval>
     isa response
     key =key
   ?manual>
     state free
   =imaginal>
     isa trial
   ==>
   =imaginal>
     response =key
   +manual>
     cmd press-key
     key =key
   =goal>
     state detect-feedback)
```

Then the retrieved-a-win and retrieved-a-non-win productions need to be changed so that they retrieve a response chunk based on the key value instead of indirectly retrieving the chunk:

```
  (p retrieved-a-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
       result win
       response =response
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state respond)

  (p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
      - result win
       response =response
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

After making that change, we should look at the model to make sure that there are not any other changes that should be made while we are adjusting it since we are going to have to retest it without production compilation before continuing to make sure it still works and making other changes now may save us from having to come back and test it without compilation yet again later.

One thing to notice is that since we now have the key to press in the trial chunks the model does not have to retrieve the response chunk in retrieved-a-win to be able to perform the key press. Similarly, retrieved-a-non-win does not need to retrieve the current response either since it could just retrieve a different response the way that guess-other does now and guess-other could be eliminated from the model. If we were not using production compilation those might be useful changes to make to the model, but production compilation should eliminate those retrievals from the model over time anyway so for now we will not make those changes to the model.

Looking at the encoding productions, encode-num-1 and encode-num-2, we see that the number chunks are also referenced by name for the trial encoding. If we go back and look at our first run with compilation turned on we can see that production0 and production1 which the model learned in fact also have references to specific chunks, three-0 and zero-0 respectively, and as we saw with production3 that means it is not going to be able to recreate and strengthen those productions. If we want to see the model response times decrease through eliminating the retrievals in that portion of the task we are also going to have to change how the model encodes the number chunks. In this case we need to have chunks in the num1 and num2 slots of the trial so that the similarities between those slot contents and the requested values will allow the model to retrieve a "close" trial chunk through partial matching when it does not have a perfectly matching trial chunk to retrieve. Thus, we cannot use the same change we did with the response chunks and just use the value of the visual-rep slot from the numbers in the trial chunks. Unlike the response chunks however the model will not need to retrieve the number chunks using the value from the slots of the trial chunk. Therefore we will not have the problem of a direct retrieval being necessary and all we need to do is provide a way for the model to reference the number chunks during the initial encoding without using the name of the chunk currently in the **retrieval** buffer.

That means that we will need to add an additional slot to the number chunk-type to hold the reference we want to use. We will call that slot representation and make this change to the chunk-type specification in the model:

```
(chunk-type number visual-rep representation)
```

That will then require making the following changes to the encoding productions to use that slot's value instead of the chunk in the **retrieval** buffer:

```
(p encode-num-1
    =goal>
      isa task
      state encode-num-1
    =retrieval>
      isa number
      representation =number
    =imaginal>
      isa trial
    ==>
    =imaginal>
      num1 =number
    =goal>
      state find-num-2
    +visual-location>
      isa visual-location
      > screen-x current
      :attended nil)

(p encode-num-2
    =goal>
      isa task
      state encode-num-2
    =retrieval>
      isa number
      representation =number
    =imaginal>
      isa trial
    ==>
    =imaginal>
      num2 =number
    =goal>
      state retrieve-past-trial)
```

Now we have to determine what value to store in that slot. It has to be a chunk so that similarities can be set, and there are basically two ways to handle that. One is to simply store the name of the number chunk itself in the representation slot when it is created. That would look like this in the current model:

```
(add-dm (zero isa number visual-rep "0" representation zero)
        (one isa number visual-rep "1" representation one)
        (two isa number visual-rep "2" representation two)
        (three isa number visual-rep "3" representation three))
```

The other option would be to create a more distributed representation which involves separate chunks for the visual mapping and the number itself. That might look something like this in the current model (though there are many ways to accomplish that):

```
(chunk-type number value)
(chunk-type number-visual visual-rep representation)

(add-dm (zero isa number value 0)
        (one isa number value 1)
        (two isa number value 2)
        (three isa number value 3)
        (isa number-visual visual-rep "0" representation zero)
        (isa number-visual visual-rep "1" representation one)
        (isa number-visual visual-rep "2" representation two)
        (isa number-visual visual-rep "3" representation three))
```

Note that for the number-visual chunks that perform the mapping from the visual representation to a number above there are no chunk names specified. The chunk name is optional when creating chunks and if one is not provided the system will generate a new name automatically. That reinforces the notion that the name of those chunks does not matter and only the content is important, but the downside to doing that is that it may make debugging the model more difficult since there will not be easily recognizable names in the trace or other inspection tools.

Which mechanism one chooses to use will depend on exactly what is required in the model and how one believes people encode that information. For this task we will go with the simpler single chunk representation, but you are welcome to try other alternatives and investigate the results as an additional exercise.

After making those changes, but before trying production compilation again, we should run the model without it to make sure that it still performs the task correctly. We need to remove the parameter setting which enables production compilation and also remove the seed value so that we can test it over multiple trials. Here are the results from the updated model:

```
Average Score of 10 trials
3.60 4.90 6.70 7.40 8.10 8.10 8.50 8.00 8.60 8.50 8.10 7.80 8.70 8.90 8.80 9.50 9.30 8.80 9.10 8.90
Average Response times
7.77 4.68 2.87 2.39 2.11 1.69 1.54 1.48 1.41 1.32 1.32 1.31 1.27 1.25 1.20 1.20 1.16 1.19 1.13 1.19
```

It still appears to be learning both with respect to increasing scores and decreasing response times. So we will re-enable production compilation, set the seed parameter again (so that we can recreate any issues which occur), and run it to see what happens with production compilation now. We will not include all of the trace here, but will include the details for important sections related both to the issues discussed above and any new issues which arise.

Looking at the productions learned during the initial encoding steps, like production0 and production1, we now see that they contain references to the number chunks themselves instead of the copy in the **retrieval** buffer when modifying the **imaginal** buffer:

```
(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM2 ZERO
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
)
```

In addition to that, on a later trial we see a production combining attende-num-2 and encode-num-2 that has been recreated and strengthened:

```
Production Compilation process started for ENCODE-NUM-2
  Production ATTEND-NUM-2 and ENCODE-NUM-2 are being composed.
  Recreating production PRODUCTION7
Parameters for production PRODUCTION7:
 :utility  1.315
 :u   1.999
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to ENCODE-NUM-2.
```

Thus, those changes to the model seem to have achieved their desired effects. Similarly, we now see production3 looking like this:

```
(P PRODUCTION3
  "NO-PAST-TRIAL & RESPOND - RESPONSE-2"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   ?MANUAL>
       STATE FREE
   ?RETRIEVAL>
       BUFFER FAILURE
 ==>
   =IMAGINAL>
       RESPONSE "d"
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "d"
)
```

and on the second trial we see that it is now also recreated:

```
Production Compilation process started for RESPOND
```

```
  Production NO-PAST-TRIAL and RESPOND are being composed.
  Recreating production PRODUCTION3
Parameters for production PRODUCTION3:
 :utility  2.376
 :u   2.857
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RESPOND.
```

Running the model until we see it successfully retrieve a past trial shows the following in the trace:

```
    38.932   DECLARATIVE            SET-BUFFER-CHUNK RETRIEVAL CHUNK4-0
    38.982   PROCEDURAL             PRODUCTION-FIRED RETRIEVED-A-WIN
Production Compilation process started for RETRIEVED-A-WIN
  Cannot compile RETRIEVE-PAST-TRIAL and RETRIEVED-A-WIN because the time between them
exceeds the threshold time.
  Setting previous production to RETRIEVED-A-WIN.
```

Previously we saw that retrieved-a-win was not valid for compilation, but now it is saying that the threshold time has been exceeded. That means the production is valid for composition, but too much time passed between the previous production's firing and the firing of this production. That happened because the retrieval took around 5 seconds to complete. Whether or not this is a problem, like many such issues, depends on one's hypothesis for what is happening when people learn in such tasks, and there are potentially multiple issues involved here. The first is whether or not one considers a 5 second retrieval to be reasonable for this task. If not, then one may want to adjust the declarative memory parameters to change that. Without data for comparison we are going to just assume that that retrieval is acceptable. Then, if one assumes that the retrieval time is acceptable, the next issue is whether one believes that the declarative knowledge must be strengthened prior to its being composed into procedural knowledge (have an activation value sufficient for it to be retrieved within the compilation threshold time) or whether production compilation should start compiling the knowledge immediately. The default setting for the production compilation threshold time is three seconds, but that value is just a conservative starting point for the system and not a strongly recommended value. For the purpose of this exercise we are going to adjust the threshold time parameter so that compilation can occur right away. To do that we must change the value of the :tt parameter to something larger than 5.224 (since that is how long the retrieval takes), and as a first pass we will choose 6 so that this pair of productions will fire. Thus, we will add this additional parameter setting to the model:

(sgp :epl t :pct t :tt 6)

After saving and reloading the model now when it gets to that point we see that it creates this production:

```
    38.982   PROCEDURAL             PRODUCTION-FIRED RETRIEVED-A-WIN
Production Compilation process started for RETRIEVED-A-WIN
  Production RETRIEVE-PAST-TRIAL and RETRIEVED-A-WIN are being composed.
  New production:

(P PRODUCTION21
  "RETRIEVE-PAST-TRIAL & RETRIEVED-A-WIN - CHUNK4-0"
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 ONE
       NUM2 THREE
 ==>
   =IMAGINAL>
   =GOAL>
       STATE RESPOND
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
Parameters for production PRODUCTION21:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVED-A-WIN.
```

That production makes the request for a particular response, "s", based on testing the specific values encoded in the trial chunk without needing to retrieve a similar trial. That is what we want to see the model do. So, now all we need to do for verifying what happens symbolically in the model is see what happens when the model retrieves a non-winning past trial. However, after running many more trials that production still does not show up in the trace as being selected and fired.

One option would be to just ignore it since it did not fire and move on to testing the model over the whole task, but perhaps it did not fire because of the particular seed value we have set for the pseudo-random number generator. We want the model to work without requiring any particular seed value being set, and that production seems like it should fire sometimes. So, before moving on we will do some more tests to see if that production ever does fire, and if so what the results from production compilation are.

One way to test this would be to just remove the seed setting and then run the model repeatedly looking at the trace each time until we find one where it fires (we would probably also want to display the starting seed each time as was shown in the unit 3 modeling text so that we can recreate the trial once we find it). In some situations doing things that way might be acceptable, but it can be a very tedious process and might not be feasible in all situations. Something that can be useful to take advantage of is that we can use the !eval! operator in the productions to actually set a flag for us so that we can write some code that will run it until that flag is set.

There are many ways one could do that, and if one is running things from the ACT-R prompt it can be a little easier since you could put Lisp code directly into that !eval!, but for consistency with what we saw earlier in the tutorial and to keep the approaches similar between the Lisp and Python implementations we will add a new ACT-R command to set the flag.

To find a game in which that production fires we will add a !eval! action like this to the production to call a command called set-flag:

```
(p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
      - result win
       response =response
     ==>
     !eval! ("set-flag")
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

We also need to remove the :seed parameter setting from the model and just add a check of the parameter at the top of the model:

```
    (sgp :seed)
```

We also want to make sure that the :v parameter is set to nil to disable the trace. Then we need to add the set-flag command and write some code to actually run the model and test that result to find a game when it happens. Here is some Lisp code evaluated at the prompt to do so:

```
? (defvar *used* nil)
*USED*
? (defun set-flag () (setf *used* t))
SET-FLAG
?  (add-act-r-command  "set-flag"  'set-flag  "Command  for  testing  the  pcomp-issues
retrieved-a-non-win production.")
T
"set-flag"
? (while (null *used*)
    (pcomp-issues-game 1))
```

Here is some similar Python code to do the same thing:

```
>>> used = False
>>> def set_flag ():
...   global used
...   used = True
...
>>> actr.add_command("set-flag",set_flag,"Command for testing the pcomp-issues re
trieved-a-non-win production.")
True
>>> while used == False:
...   pcomp_issues.game(1)
...
```

When you use either of those you will see some output like this that prints the seed value and then the model results repeatedly until a game is found where it gets used:

```
:SEED (79246602991 0) (default NO-DEFAULT) : Current seed of the random number g
enerator
Average Score of 1 trials
3.00 9.00 8.00 6.00 8.00 6.00 7.00 7.00 7.00 9.00 9.00 9.00 8.00 9.00 7.00 6.00 10.00 6.00 10.00 10.00
Average Response times
9.72 3.92 1.83 2.03 1.54 1.73 1.44 1.47 1.43 1.17 1.34 1.24 1.19 1.11 1.28 1.33 1.01 1.29 1.01 1.03
:SEED (79246602991 9581) (default NO-DEFAULT) : Current seed of the random number generator
Average Score of 1 trials
5.00 5.00 8.00 4.00 8.00 8.00 8.00 4.00 10.0 10.0 10.0 10.0 10.0 10.0 9.00 10.0 10.0 10.0 8.00 10.0
Average Response times
8.09 2.67 1.78 1.44 1.80 1.66 1.57 1.31 1.35 1.16 1.17 1.09 1.02 1.27 1.10 1.09 1.11 1.08 1.00 0.95
...
:SEED (79246602991 77656) (default NO-DEFAULT) : Current seed of the random number generator
Average Score of 1 trials
```

```
-3.00 8.00 10.0 9.00 9.00 10.0 10.0 10.0 9.00 9.00 10.0 8.00 10.00 9.00 10.00 10.00 10.00 9.00 10.00 7.00
Average Response times
8.64 3.42 1.99 2.13 1.60 1.27 1.13 1.31 1.28 1.20 1.10 1.28 1.01 1.06 1.14 0.99 1.03 1.10 0.99 1.06
```

If you try that you will see different seed values displayed, but eventually it should stop and the last seed value shown will result in a game where that production fires. Before looking at the trace of that trail we will first remove that !eval! from the production because that will cause problems for production compilation with a warning like this:

```
Production Compilation process started for RETRIEVED-A-NON-WIN
  Production RETRIEVED-A-NON-WIN is not valid for compilation
   because it contains one or more !eval! operators
```

Since the procedural system does not know what happens because of that external call through !eval! it considers it unsafe to compose that production.

We then need to set the seed to (79246602991 77656) since that is the value we found above and turn the trace back on. Then we will run it a trial at a time to find where that production fires.

```
Production Compilation process started for RETRIEVED-A-NON-WIN
  Production RETRIEVED-A-NON-WIN is not valid for compilation
   because it has conditions with modifiers on slot tests
```

It indicates that the production is not valid for compilation because it has modifiers on the slot tests. Here is the production again for reference:

```
  (p retrieved-a-non-win
     =goal>
       isa task
       state process-past-trial
     =retrieval>
       isa trial
      - result win
       response =response
     ==>
     +retrieval>
       isa response
       key =response
     =goal>
       state guess-other)
```

The slot test highlighted above is the only one that has a modifier so that must be what is stopping compilation.

That is an important issue to keep in mind when working with production compilation -- it cannot compile productions which have tests for inequalities for reasons similar to not composing across a retrieval failure – one does not generally want to encode that something was not true and assume that will always be the case. However it is often convenient to have such tests in the productions which one wants to be compiled. There are a couple of ways to handle that. The first is to replace the production with one or more productions that perform the same calculation using a positive test. In this case that would mean adding retrieved-a-lose and retrieved-a-draw productions which test for those values explicitly as retrieved-a-win does. Since there are only three possible options that would not be a difficult change to make for the model, but in other situations that might not be feasible because there may be too many choices or not all the possibilities may be known in advance. An alternative, which we will use here, is to just bind the value from the slot to a variable in the buffer test and then perform the inequality test in code. Although we noted above that a !eval! blocks the composition, there is a special version which allows one to tell the procedural system that you are guaranteeing the external code to be “safe” with respect to production compilation. To do that you can use the !safe-eval! operator instead. That could go out through an external command, as we used above, but for simplicity we are just going to perform the check directly in Lisp code in the production:

```
(p retrieved-a-non-win
   =goal>
     isa task
     state process-past-trial
   =retrieval>
     isa trial
     result =result
     response =response
   !safe-eval! (not (equal =result 'win))
   ==>
   +retrieval>
     isa response
     key =response
   =goal>
     state guess-other)
```

If we save that change and run the model to that point again we will now see the following production compilation trace result:

```
Production Compilation process started for RETRIEVED-A-NON-WIN
  Production RETRIEVE-PAST-TRIAL and RETRIEVED-A-NON-WIN are being composed.
  New production:

(P PRODUCTION45
  "RETRIEVE-PAST-TRIAL & RETRIEVED-A-NON-WIN - CHUNK12-0"
   !SAFE-EVAL! (NOT (EQUAL (QUOTE LOSE) (QUOTE WIN)))
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 THREE
       NUM2 THREE
 ==>
   =IMAGINAL>
   =GOAL>
       STATE GUESS-OTHER
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
Parameters for production PRODUCTION45:
 :utility    NIL
 :u   0.000
 :at  0.050
 :reward    NIL
 :fixed-utility    NIL
  Setting previous production to RETRIEVED-A-NON-WIN.
```

This time it created a production which will retrieve the response for "s" whenever it has encoded a trial of the numbers three and three. The !safe-eval! from the retrieved-a-non-win production has been included in the conditions of this production, but because the retrieval chunk's contents were compiled into the production the test is now explicitly testing that the symbol lose is not equal to the symbol win which will always be true. Unlike the compilation of retrieve-past-trial and retrieved-a-win however this production is not actually mapping a specific trial to a particular result because the production which fires after retrieved-a-non-win, guess-other, will retrieve a different response to make since the model does not want to make the response that did not lead to a win:

```
(p guess-other
     =goal>
       isa task
       state guess-other
     =retrieval>
       isa response
       key =key
     ==>
     +retrieval>
       isa response
      - key =key
     =goal>
       state respond)
```

Therefore when it retrieves a non-winning trial it is not going to immediately create a new production which performs a specific move. Since retrieved-a-non-win does not seem to fire very often (we had to search to find a game in which it did) that is not likely to be an issue in the model, but it is worth keeping in mind for any analysis we do later.

Before moving on to looking at the performance there is one last detail to mention. The guess-other production shown above includes a negative modifier in its request to the **retrieval** buffer so that it will retrieve a response which does not match the current one. Unlike inequality tests in the conditions however modifiers in a request are allowed for production compilation and we see this production as the result of production compilation for retrieved-a-non-win and guess-other in the trace and it keeps the modifier in the request:

```
Production Compilation process started for GUESS-OTHER
  Production RETRIEVED-A-NON-WIN and GUESS-OTHER are being composed.
  New production:

(P PRODUCTION46
  "RETRIEVED-A-NON-WIN & GUESS-OTHER - RESPONSE-1"
   !SAFE-EVAL! (NOT (EQUAL =RESULT (QUOTE WIN)))
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =RETRIEVAL>
       RESPONSE "s"
       RESULT =RESULT
 ==>
   =GOAL>
       STATE RESPOND
   +RETRIEVAL>
    -  KEY "s"
       IS-RESPONSE T
)
```

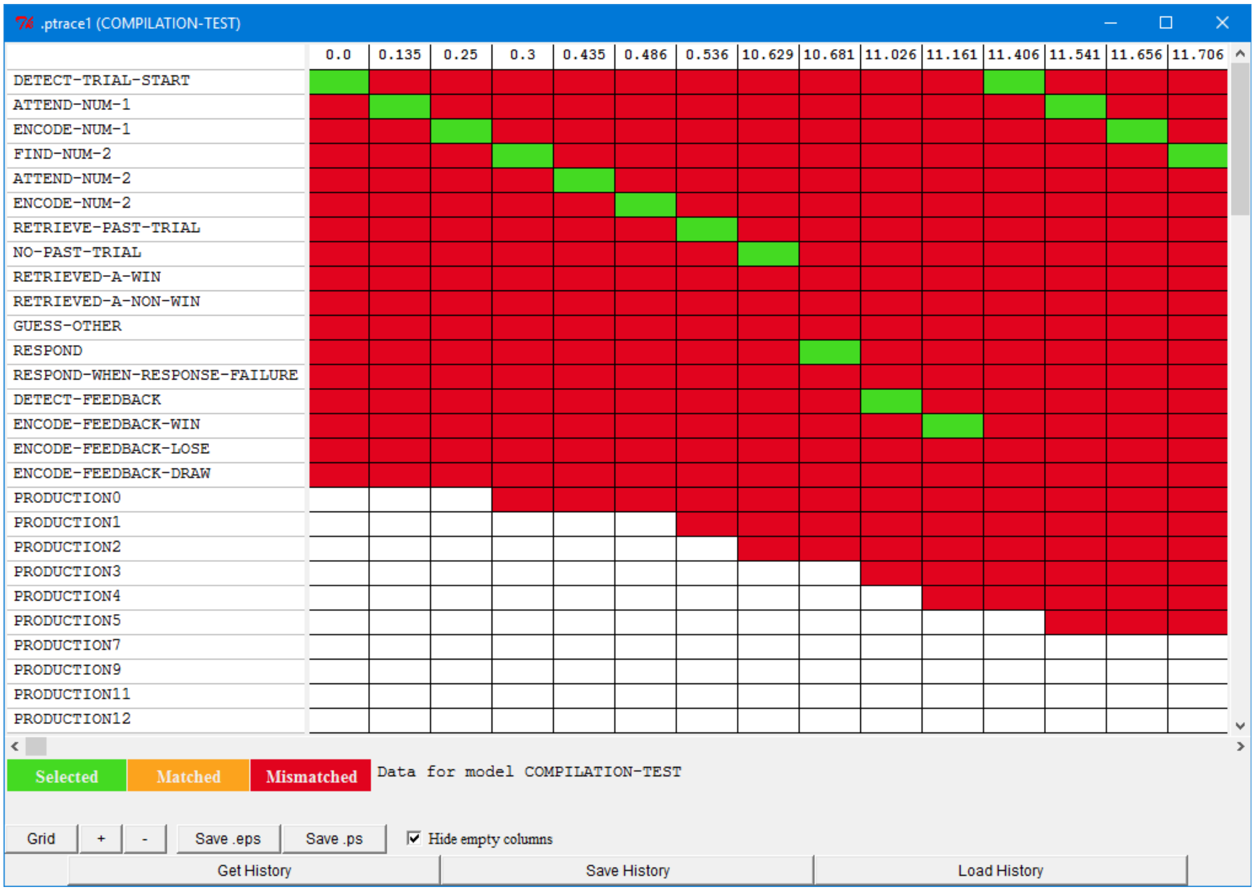
The reason it can keep a modifier in a request is because that request is not a constraint of the procedural system – all the information which is contained in a request is either constant values or was truthfully bound in the condition of the production. Therefore, it is not encoding any assumption that something is false or not available when composing that request.

Now we have verified that production compilation is able to compose the starting productions from the task into productions that seem reasonable. The next thing to investigate is whether or not the compiled productions are being used by the model and if so whether they are having an effect on how it performs the task.

There are many ways one can look for that, but here we will show how the "Production" grid tool in the Environment can be useful with production compilation. As we did above, we need to open the tool before running the model and then press the "Get History" button once the model is done. We will leave the seed value set to (79246602991 77656) so that we can recreate this run if we want to look at it again in detail, but turn the :v parameter off again since we don't need to see the trace information. Here is the result from one game with that seed:

```
Average Score of 1 trials
-3.00 8.00 7.00 6.00 8.00 8.00 10.0 10.0 10.0 10.0 8.00 10.00 10.0 8.00 10.0 8.00 8.00 10.0 10.0 10.0
Average Response times
8.64 3.49 2.67 1.55 1.73 1.42 1.58 1.47 1.38 1.13 1.14 1.09 1.13 1.21 1.15 1.09 1.02 1.10 1.08 1.06
```

Looking at the results displayed the model still seems to be performing the task correctly and is still getting faster and more accurate as it performs the task. So there do not appear to be any problems introduced because of the productions that are being composed. In the production grid it is going to take a little while for the display to update after hitting "Get History" because of the amount of data to display, but once it does there should be a lot of new productions listed and many columns of data. It may help to check the "Hide empty columns" box at the bottom to remove the output for conflict resolution events that did not result in selecting a production. The results will look something like this:

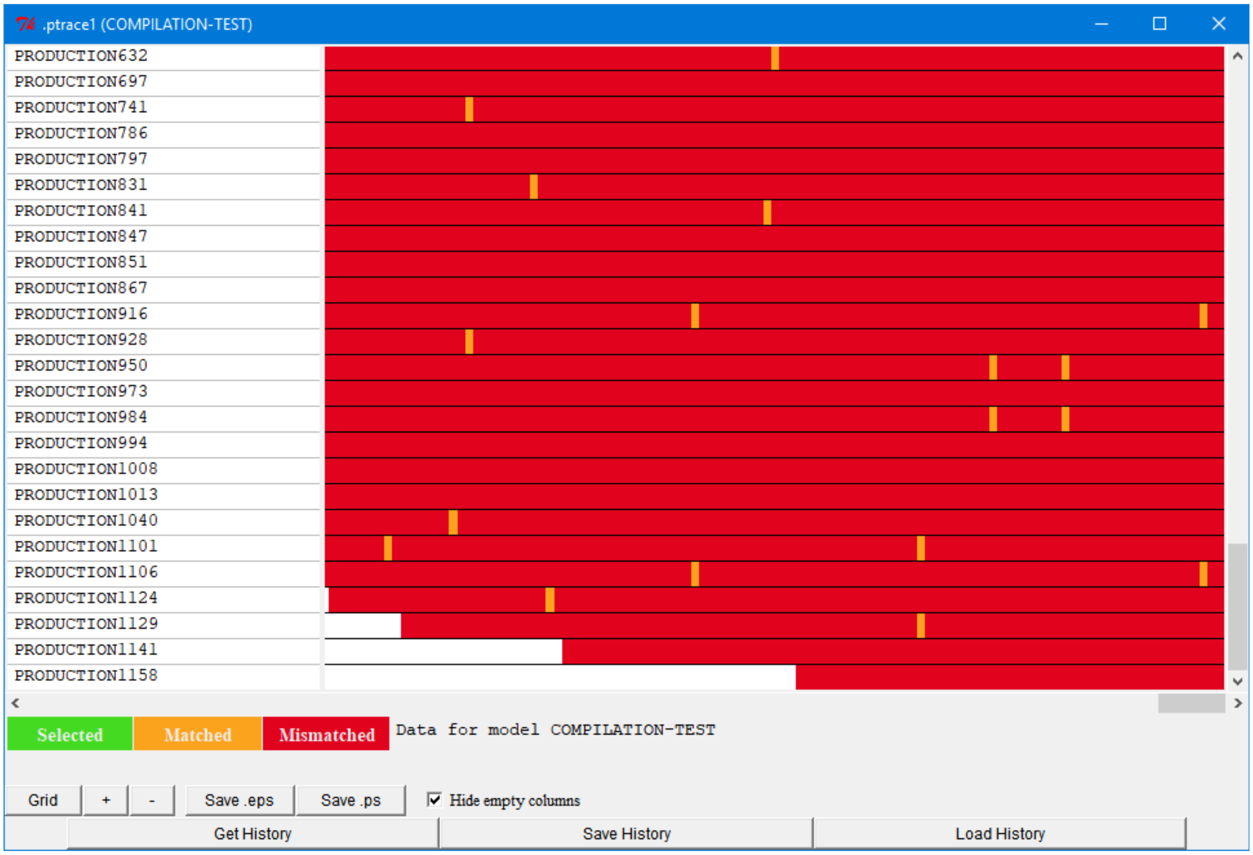


	0.0	0.135	0.25	0.3	0.435	0.486	0.536	10.629	10.681	11.026	11.161	11.406	11.541	11.656	11.706
DETECT-TRIAL-START	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥
ATTEND-NUM-1	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ENCODE-NUM-1	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
FIND-NUM-2	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩
ATTEND-NUM-2	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-NUM-2	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVE-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
NO-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-WIN	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-NON-WIN	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
GUESS-OTHER	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND-WHEN-RESPONSE-FAILURE	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
DETECT-FEEDBACK	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-WIN	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-LOSE	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-DRAW	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION0	⬜	⬜	⬜	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION1	⬜	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION2	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION3	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION4	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥	🟥	🟥
PRODUCTION5	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥
PRODUCTION7	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
PRODUCTION9	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
PRODUCTION11	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
PRODUCTION12	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜

```
Data for model COMPILATION-TEST
```

We discussed how to read the results of this display previously, but there is something new about this display because of production compilation. The newly compiled productions have white boxes in some columns which do not report any details when the mouse is placed over them. Those boxes indicate that that production did not exist at that time. Thus the first non-white box in a row indicates approximately when the production was created because that was the first time it was attempted to be selected.

The composed productions are also displayed in the order in which they were created. This provides us with a fairly easy way to determine if the model is continuing to learn new productions throughout the task, or if there appears to be a point at which it has learned all the new productions that it can. If we zoom out on the display by hitting the "-" button, turn off the vertical lines by hitting the "Grid" button, scroll down to the last new production, and then scroll right to see the end of the task we will see something like this:



PRODUCTION632
PRODUCTION697
PRODUCTION741
PRODUCTION786
PRODUCTION797
PRODUCTION831
PRODUCTION841
PRODUCTION847
PRODUCTION851
PRODUCTION867
PRODUCTION916
PRODUCTION928
PRODUCTION950
PRODUCTION973
PRODUCTION984
PRODUCTION994
PRODUCTION1008
PRODUCTION1013
PRODUCTION1040
PRODUCTION1101
PRODUCTION1106
PRODUCTION1124
PRODUCTION1129
PRODUCTION1141
PRODUCTION1158

```
Data for model COMPILATION-TEST
```

That shows that even near the end of the task this model was still composing new productions. That may or may not be a good thing depending on what one was expecting for the task. Given the overall length of our task, approximately 10 minutes, it does not seem unreasonable that there are still opportunities for further learning at the end, but in other models one might expect compilation to slow down or stop before the end of the task.

Now we will start looking at the productions which the model has generated in more detail. If there were not as many then it might be worthwhile to use the "Procedural" viewer to look at all of them to see what they look like and what their utilities are at the end. However, since there are more than 100 composed productions and there did not appear to be any problems as it performed the task we are going to just look for productions that have an interesting history to investigate. In particular, the things that will be considered interesting are productions which never match because those might indicate a problem which we did not notice previously and new productions which are actually used by the model because those should be the ones that we are expecting it to learn and use.

There are a few ways to find productions which are never matched based on the details recorded automatically by ACT-R. One way is by using the Grid tool in the Environment and looking for rows with no orange or green boxes in them. If we zoom out they should be fairly easy to locate, and some of the first few productions learned, production0, production4, and production5 all seem to have that property, as do several others. Another way to find them would be to use the "Procedural" viewer to look for productions which have a :utility parameter value of nil. That parameter records the utility the production had the last time it matched, and if it is nil it means that it has never matched. We can also test that parameter value in code because we can get the production parameters using the spp command. That allows us to do something like this in Lisp to create a list of all the productions which have a nil :utility parameter setting:

```
? (mapcar 'car (remove-if (lambda (x) x) (no-output (spp :name :utility)) :key 'second))
(RESPOND-WHEN-RESPONSE-FAILURE  PRODUCTION0  PRODUCTION4  PRODUCTION5  PRODUCTION6  PRODUCTION11
PRODUCTION12  PRODUCTION22  PRODUCTION29  PRODUCTION46  PRODUCTION47  PRODUCTION189  PRODUCTION242
PRODUCTION847   PRODUCTION867   PRODUCTION973   PRODUCTION994   PRODUCTION1008   PRODUCTION1141
PRODUCTION1158)
```

or something very crudely like this in Python (I'm sure there are much nicer ways to do so):

```
>>> actr.hide_output()
>>> all = actr.spp(':name',':utility')
>>> actr.unhide_output()
>>> for i in all:
...   if i[1] == None:
...     print(i[0])
...
RESPOND-WHEN-RESPONSE-FAILURE
PRODUCTION0
PRODUCTION4
PRODUCTION5
PRODUCTION6
PRODUCTION11
PRODUCTION12
PRODUCTION22
PRODUCTION29
PRODUCTION46
PRODUCTION47
PRODUCTION189
PRODUCTION242
PRODUCTION847
PRODUCTION867
PRODUCTION973
PRODUCTION994
PRODUCTION1008
PRODUCTION1141
PRODUCTION1158
>>>
```

However we go about finding them, there are 20 such productions in this model. We will not look at each individually here, but what you will find if you do is that they basically fall into four general categories which we will discuss. Before continuing, you might want to look them over and see if you can find the similarities among them yourself.

The first category are those that we already knew would not be used -- productions which are composed from a production which makes a response and one which detects the result of that response. Those involve either detect-feedback or detect-trial-start as the second production in the pair. Since we expected these to occur we do not need to investigate them further.

The next category are productions for rare situations, particularly those dealing with the retrieved-a-non-win production. We know that is not a common occurrence in the model since we had to search to find a game in which it occurred. Because of that the productions composed from it are also not likely to have an opportunity to match either. That does not seem to be a problem we need to investigate any further. One of the productions which never matches is actually starting production in the model: respond-when-response-failure. The respond-when-response-failure production is only needed if the model ever fails to retrieve a response, and since that should not happen we would not expect to see that production selected and fired. It could probably be removed from the starting model without affecting things, but it is often safest to include productions like that in a model so that it can deal with unexpected situation. It is possible, no matter how unlikely, for the noise in the activations to push all chunks below the retrieval threshold and if the model does not have any productions to deal with failures to retrieve it will be stuck and unable to perform the task.

Another category of productions which does not match are those created late in the run which have very specific constraints. Presumably those productions are not matching because that specific pair of numbers is not presented again before the end of the experiment. Here are some examples of that:

```
(P PRODUCTION973
  "PRODUCTION19 & RETRIEVED-A-WIN - CHUNK74-0"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
       NUM1 ZERO
   =VISUAL>
       VALUE "1"
 ==>
   =IMAGINAL>
       NUM2 ONE
   =GOAL>
       STATE RESPOND
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
(P PRODUCTION1158
  "RETRIEVE-PAST-TRIAL & PRODUCTION26 - CHUNK92-0"
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 ZERO
       NUM2 TWO
   ?MANUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE "s"
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "s"
)
```

Production1158 is the type of production that we were expecting the model to learn. It maps a specific **imaginal** chunk representation to a specific action. Production973 seems to be a further step in that process where the encoding of the number from the **visual** buffer into the **imaginal** is also composed with the response. Seeing these productions is a good sign because they are what we expected and are composed from previously composed productions so that means the model is actually using some of the composed productions which we will look into further shortly.

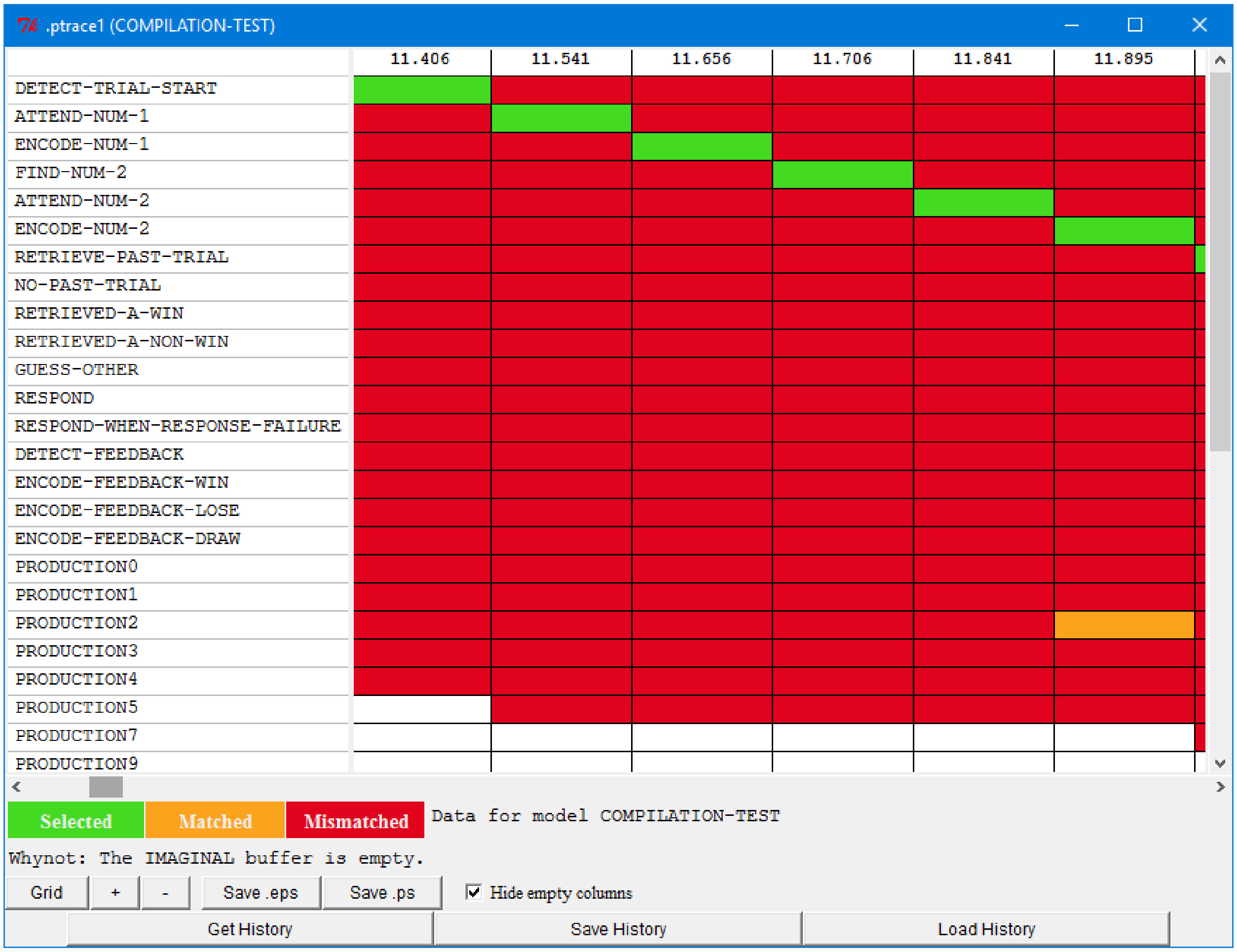
The final category of productions which are not being matched are productions composed from attend-num-1 and encode-num-1. There are four such productions, one for each of the numbers retrieved (zero, one, two, and three). They all have the same structure and here is one of them for reference:

```
(P PRODUCTION0
  "ATTEND-NUM-1 & ENCODE-NUM-1 - ZERO"
   =GOAL>
       STATE ATTEND-NUM-1
   =IMAGINAL>
   =VISUAL>
       VALUE "0"
 ==>
   =IMAGINAL>
       NUM1 ZERO
   =GOAL>
       STATE FIND-NUM-2
   +VISUAL-LOCATION>
       :ATTENDED NIL
    >  SCREEN-X CURRENT
)
```

We discussed this production before and expected it to help the model speed up over time, so the question is why isn't it being selected? If we look for the similar productions which compose attend-num-2 and encode-num-2, like production1:

```
(P PRODUCTION1
  "ATTEND-NUM-2 & ENCODE-NUM-2 - ONE"
   =GOAL>
       STATE ATTEND-NUM-2
   =IMAGINAL>
   =VISUAL>
       VALUE "1"
 ==>
   =IMAGINAL>
       NUM2 ONE
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
)
```

we find that it is matched multiple times over the course of the experiment so it seems odd that production0 is not also matched. To figure out why production0 is not matched we can use the Grid tool to look at the why-not information for production0 when we would expect it to be matched, which is when attend-num-1 matches since that is the parent production with which it should be competing. To find that it is probably easiest to zoom in again and restore the grid lines. The first column that we find which has attend-num-1 selected while production0 exists is at time 11.541 and this is what we find when we place the cursor over the red box in the production0 row:



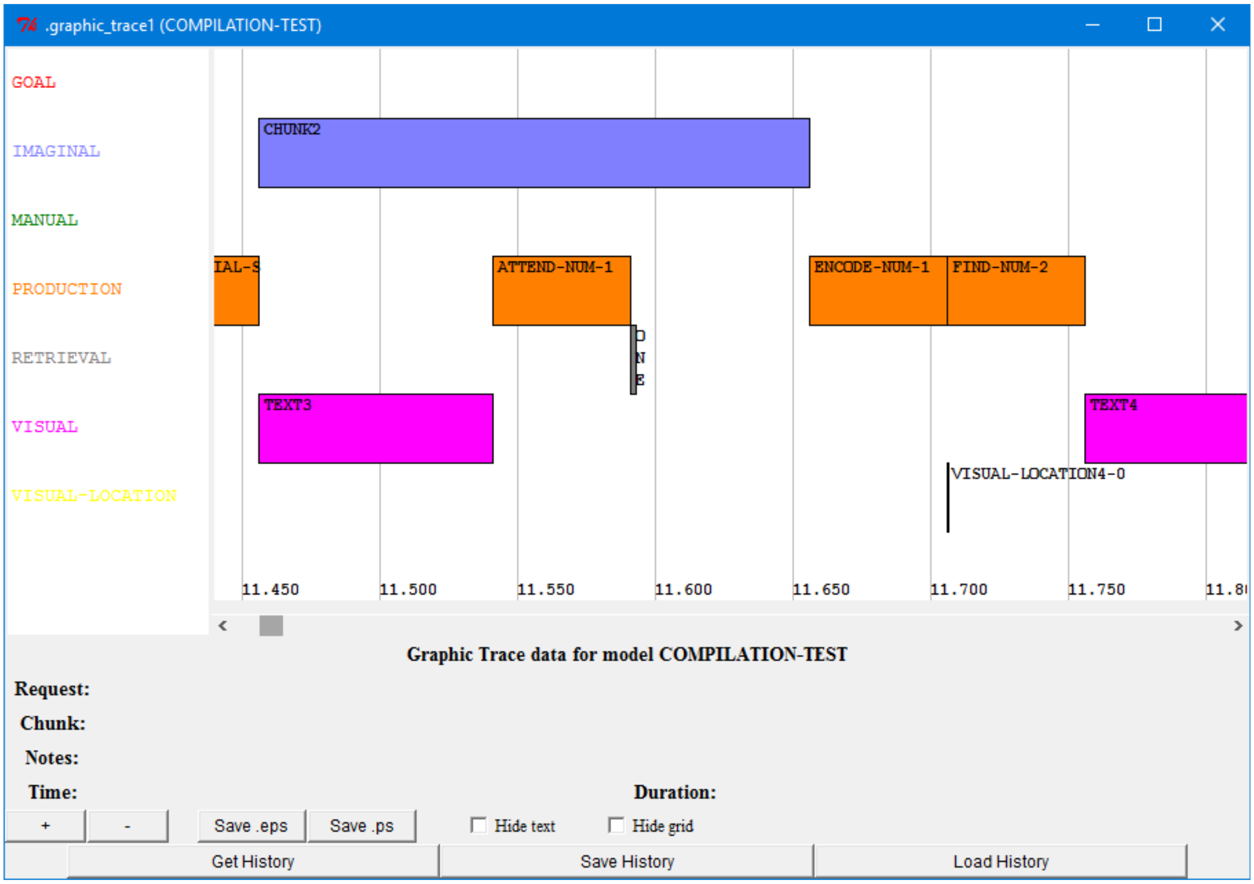
	11.406	11.541	11.656	11.706	11.841	11.895
DETECT-TRIAL-START	🟩	🟥	🟥	🟥	🟥	🟥
ATTEND-NUM-1	🟥	🟩	🟥	🟥	🟥	🟥
ENCODE-NUM-1	🟥	🟥	🟩	🟥	🟥	🟥
FIND-NUM-2	🟥	🟥	🟥	🟩	🟥	🟥
ATTEND-NUM-2	🟥	🟥	🟥	🟥	🟩	🟥
ENCODE-NUM-2	🟥	🟥	🟥	🟥	🟥	🟩
RETRIEVE-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥
NO-PAST-TRIAL	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-WIN	🟥	🟥	🟥	🟥	🟥	🟥
RETRIEVED-A-NON-WIN	🟥	🟥	🟥	🟥	🟥	🟥
GUESS-OTHER	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND	🟥	🟥	🟥	🟥	🟥	🟥
RESPOND-WHEN-RESPONSE-FAILURE	🟥	🟥	🟥	🟥	🟥	🟥
DETECT-FEEDBACK	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-WIN	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-LOSE	🟥	🟥	🟥	🟥	🟥	🟥
ENCODE-FEEDBACK-DRAW	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION0	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION1	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION2	🟥	🟥	🟥	🟥	🟥	🟧
PRODUCTION3	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION4	🟥	🟥	🟥	🟥	🟥	🟥
PRODUCTION5		🟥	🟥	🟥	🟥	🟥
PRODUCTION7						
PRODUCTION9						

```
Data for model COMPILATION-TEST
Whynot: The IMAGINAL buffer is empty.
```

It is not matching because the **imaginal** buffer is empty. The question then becomes why is the **imaginal** buffer empty at that time? If you look at the model's productions you may be able to ascertain why that is happening, but if not there are multiple ways to look into that further. One would of course be to run it again with the trace on and look at the trace to see if you can determine why. Another option would be to step through the operation with the Stepper tool so that you can inspect things more closely as they occur. Something else which can be done, and which we will use here, is to use a "Graphic trace" tool for recorded data from the Environment instead of the text trace to try to determine what is happening.

To do that we need enable the saving of the "Graphic trace" information by opening the tool before we run the model (either the horizontal or vertical version can be used and we'll show the horizontal one here). Since we know this happens on the second trail we can just run the model for two trials instead of waiting for it to run the entire experiment.

Once that's done we can press the "Get History" button and then go to the appropriate time of that happening:



Graphic Trace data for model COMPILATION-TEST

Request:

Chunk:

Notes:

Time:

Duration:

There we see that the **imaginal** module is busy creating a chunk at time 11.541 as requested by the detect-trial-start production, and attend-num-1 is selected while that request is ongoing. Production0, like encode-num-1 which it is composed from, requires that there be a chunk in the **imaginal** buffer to match. Since attend-num-1 does not have that requirement it can be selected while the imaginal module is still busy. That is another important thing to remember about production compilation - a composed production will have to meet the constraints imposed by both parents. If, as is the case here, the constraints for the second production take time to occur then that composed production may not compete with its first parent and may never match. While it seems like this is a lost opportunity for speedup in the model, looking at the other information in the graphic trace actually shows that it does not really matter. That is because the retrieval of the number chunk also completes before the **imaginal** chunk is created. Thus, the time spent creating that **imaginal** chunk determines when encode-num-1 (or our composed production0) will be able to be selected and fired. Eliminating attend-num-1 and the number retrieval through composition would not change that timing. If we wanted to see a speedup from composing these productions we would have to adjust when the model makes the request to create the **imaginal** chunk so that it does not dominate this timing or change the time it takes for **imaginal** actions to occur. That does not seem like something worth changing in the model since we are primarily expecting the speedup to occur because of composing the specific response information in this model, but you are welcome to try those options as an additional exercise if you like.

Now that we have looked at the composed productions which are not matching we will look at those which are being selected and fired to make sure that the model is learning to use the new productions that we expected. Like finding those that were not matched there are multiple options available for finding those which do match. However, there is not a simple parameter or other automatically recorded information which we can test to do so. Thus, getting this information will require either using the production grid tool or setting additional parameters in the model before running it. Probably the easiest way is to again use the "Production" grid, and this time instead of looking for empty rows we are looking for rows with lots of green and orange in them. If we want to see which productions are selected we can get that from the model trace if we enable it, but to see those that match but which are not selected we will also have to enable either the :cst or :crt parameter to include the additional conflict resolution information. If we want to collect that information in a list or process it in code then we would have to explicitly collect the information while the model runs using the :conflict-set-hook parameter or ask the module to record the history internally and then get the data when it is done. How to use the conflict-set-hook and the history recording tools are beyond the scope of this document, but details can be found in the reference manual.

Looking at the history there are lots of new productions which are matching frequently, but there are only a few which are getting selected and fired frequently. Those productions are production2, production7, production26, production57, production86, production226, and production377. Those productions seem to fall into two categories: productions which are collapsing the steps needed for encoding the second item and productions which are making a response based on a retrieved response chunk. We expected items of the first type to be created and used, but the second type, like production26, are not quite what we were looking for:

```
 (P PRODUCTION26
  "RETRIEVED-A-WIN & RESPOND - RESPONSE-1"
   =GOAL>
       STATE PROCESS-PAST-TRIAL
   =IMAGINAL>
   =RETRIEVAL>
       RESPONSE "s"
       RESULT WIN
   ?MANUAL>
       STATE FREE
 ==>
   =IMAGINAL>
       RESPONSE "s"
   =GOAL>
       STATE DETECT-FEEDBACK
   +MANUAL>
       CMD PRESS-KEY
       KEY "s"
)
```

That production has removed a retrieval which should reduce the time it takes to respond, but it is not the main type of production we were looking to create. That production does not map the trial information to a particular response. It just eliminates the retrieval of the response chunk that occurred before it made the response. The productions we really want the model to start using will be a combination of retrieve-past-trial and retrieved-a-win or another production which has been composed from retrieved-a-win. So, now we will look for some of those and see why they are not being selected.

Looking through the generated productions we do find instances of the productions we want, like production45:

```
(P PRODUCTION45
  "RETRIEVE-PAST-TRIAL & RETRIEVED-A-NON-WIN - CHUNK12-0"
   !SAFE-EVAL! (NOT (EQUAL (QUOTE LOSE) (QUOTE WIN)))
   =GOAL>
       STATE RETRIEVE-PAST-TRIAL
   =IMAGINAL>
       NUM1 THREE
       NUM2 THREE
 ==>
   =IMAGINAL>
   =GOAL>
       STATE GUESS-OTHER
   +RETRIEVAL>
       IS-RESPONSE T
       KEY "s"
)
```

Looking at the history shows productions like that do match a few times, but they are not being recreated enough to raise their utilities to a point where they are able to be selected over the original productions. Since it is creating them and they do match, that is all we are concerned with for now.

Now that we have looked at the productions the model learns and seen that they do not cause any problems for the model's ability to do the task we can start looking at the effect they have on the model's performance on the task. To do that we will want to remove the seed setting from the model and run it over multiple trials to see the average results. When doing that it will also be a good idea to look at the individual game outcomes as well to make sure there are not any problems along the way, and printing the seed for each will allow us to recreate a bad run if we see one.

To help with that we can use the optional parameter of the pcomp-issues-game function in Lisp and the game function in the pcomp_issues module for Python to have it output the results for each game run before displaying the average at the end. Here are some results from running 10 games with the individual game results and each game's starting seed shown:

```
:SEED (5775195006 238051) (default NO-DEFAULT) : Current seed of the random number generator
Score
   4  10   9  10   7   9  10  10   9  10  10  10  10  10  10  10  10  10  10  10
Average response times
6.82 2.44 1.93 2.45 1.45 1.38 1.31 1.29 1.41 1.22 1.10 1.16 1.19 1.05 1.13 1.10 1.04 1.10 0.95 1.09
:SEED (5775195006 248483) (default NO-DEFAULT) : Current seed of the random number generator
Score
   3  10   7   7   9  10  10  10   9  10  10  10  10  10  10  10   9   7  10  10
Average response times
8.76 3.40 1.96 1.87 1.33 1.45 1.29 1.34 1.20 1.37 1.11 1.14 1.07 1.17 1.14 1.03 1.03 1.13 1.01 1.04
:SEED (5775195006 258658) (default NO-DEFAULT) : Current seed of the random number generator
Score
  -1  -2  -2   8  10   4   8   8  10   8  10  10   8  10  10  10  10  10  10   8
Average response times
6.56 5.86 5.59 2.82 1.84 1.79 1.56 1.55 1.31 1.28 1.20 1.11 1.17 1.35 1.19 1.05 1.15 0.97 1.05 0.99
:SEED (5775195006 269418) (default NO-DEFAULT) : Current seed of the random number generator
Score
   1   3   6   6   8   7   8   6   4  10   6   8   8  10   8   8   8   6  10  10
Average response times
8.35 6.48 3.22 2.27 1.52 2.22 2.02 1.43 1.57 1.42 1.31 1.37 1.12 1.16 1.36 1.18 1.11 1.08 1.01 1.05
:SEED (5775195006 279707) (default NO-DEFAULT) : Current seed of the random number generator
Score
   3   3   9   6   8   8  10   8   9  10  10   9  10  10  10  10  10  10  10  10
Average response times
```

```
8.29 4.19 3.51 1.83 2.27 1.19 1.16 1.37 1.25 1.33 1.16 1.16 1.11 1.02 1.08 1.07 1.04 1.00 1.02 1.01
:SEED (5775195006 290250) (default NO-DEFAULT) : Current seed of the random number generator
Score
  -2   6   7   8  10  10  10   8  10  10  10  10  10  10  10  10  10  10  10  10
Average response times
10.02 6.12 3.14 1.97 1.57 1.55 1.34 1.64 1.15 1.25 1.14 1.05 1.22 1.12 1.16 1.05 1.09 1.01 1.08 1.00
:SEED (5775195006 301195) (default NO-DEFAULT) : Current seed of the random number generator
Score
   6  10  10   8   9   7   6   8  10   7   9   6   8   8   8   8  10   9   9  10
Average response times
5.82 3.63 1.84 1.51 1.93 1.59 1.62 1.42 1.51 1.31 1.23 1.16 1.24 1.18 1.07 1.08 1.01 1.10 1.10 1.00
:SEED (5775195006 310963) (default NO-DEFAULT) : Current seed of the random number generator
Score
   3   7   2   7   6  10   8   8   4   8   8  10   9   6   5   9  10   8  10  10
Average response times
7.45 3.85 2.91 2.87 2.64 1.93 1.38 1.42 1.39 1.48 1.51 1.16 1.12 1.45 1.17 1.21 1.32 1.06 1.12 1.03
:SEED (5775195006 321560) (default NO-DEFAULT) : Current seed of the random number generator
Score
   7   9   9   7   8  10   8   8   8   9  10   8   9  10   9  10   9  10  10   9
Average response times
8.16 2.85 1.74 1.87 1.63 1.49 1.30 1.39 1.23 1.41 1.25 1.23 1.22 1.13 1.07 1.05 1.13 1.04 1.04 1.07
:SEED (5775195006 331681) (default NO-DEFAULT) : Current seed of the random number generator
Score
  -1   9   9   8   8   9   8   8   8   7  10   8  10   7  10   9  10   9   9   8
Average response times
9.74 4.49 3.06 1.80 1.49 1.37 1.45 1.32 1.34 1.23 1.28 1.12 1.13 1.03 1.17 1.04 1.05 1.10 1.02 1.04

Average Score of 10 trials
2.30 6.50 6.60 7.50 8.30 8.40 8.60 8.20 8.10 8.90 9.30 8.90 9.20 9.10 9.00 9.40 9.60 8.90 9.80 9.50
Average Response times
8.00 4.33 2.89 2.13 1.77 1.60 1.44 1.42 1.34 1.33 1.23 1.17 1.16 1.17 1.16 1.09 1.10 1.06 1.04 1.03
```

The average results still show the same learning patterns we expect, the scores go up and response time goes down, and the individual games do not seem to show any particularly unusual situations occurring. We could run some more tests, but since we have inspected the productions the model learns fairly thoroughly and this small test looks good we are going to assume that it is working well and move on to looking at the average data.

Here are the results of the model without production compilation averaged over 50 runs:

```
Average Score of 50 trials
2.06 5.22 6.12 7.56 7.86 8.00 8.22 8.36 8.44 7.94 8.86 8.86 8.52 8.62 8.74 9.24 8.82 8.70 9.10 9.00
Average Response times
7.97 4.68 3.21 2.36 1.94 1.68 1.56 1.47 1.39 1.38 1.31 1.27 1.26 1.25 1.20 1.19 1.21 1.18 1.16 1.15
```

and here are the results of the model with production compilation averaged over 50 runs:

```
Average Score of 50 trials
2.88 5.78 6.76 7.52 7.90 7.76 8.10 8.16 8.88 8.00 8.40 8.78 8.42 8.72 8.98 9.02 8.82 8.44 8.46 8.88
Average Response times
7.73 4.63 2.96 2.31 1.78 1.60 1.51 1.39 1.35 1.27 1.20 1.20 1.17 1.14 1.11 1.10 1.09 1.08 1.08 1.06
```

The average scores look fairly similar between the two as do the response times, and about the only difference seems to be that the model is slightly faster with production compilation. So, unfortunately, setting up production compilation to work with that starting model has had little effect on the results. The most likely reason for the response times not being much different is because the initial model was already fairly compact in terms of the number of productions which it needed to perform the task and its use of base-level learning quickly sped up the retrievals necessary. Thus there was not a lot that compilation could remove to improve the speed. As for the scores, the effect we wanted (compiling specific response productions for the winning move on each potential trial) does not happen because as we saw above there are not enough trials for those productions to learn a utility strong enough to dominate the initial productions. Without any actual data to fit the model to there are no specific adjustments that we need to make now to adjust the model's performance, but we will describe some adjustments that could be made and you are welcome to investigate those changes or others to see what effects they have on the model's results.

If we wanted the model to show a more gradual speedup in response time through production compilation then we would have to make significant changes to the starting model so that it required more productions and more retrievals to perform the task initially. One way to do that would be to convert the model so that it has to retrieve task instructions like the unit 7 paired associate task instead of starting out with an already optimized set of task specific productions. Alternatively, we could change the declarative memory parameters that it uses so that it is not as fast to begin with, but that could also be done without the need for production compilation. Just changing the parameters for production compilation, like slowing the learning rate or adjusting the initial utilities, would not allow us to make the model perform any slower than the starting model because utility learning will favor the faster productions as long as they lead to the same rewards which they will in this task as long as the model is responding correctly.

If we want the model to speed up even more through production compilation then we could increase the utility learning rate so that the new productions get higher utilities sooner. We could also increase the noise parameter or the starting utilities of composed productions so that they are more likely to be selected and gain their own rewards sooner. That might help the model to use the productions we wanted it to learn sooner. However a change like that might also make the scores go down because it could allow composed productions which make bad responses to get selected more often as well as the good ones. As an example, here are the results from running the model with an :alpha value of .9 (a very fast learning rate):

```
Average Score of 30 trials
2.13 3.90 4.90 6.00 6.93 7.20 7.70 7.63 7.37 7.37 7.77 7.73 8.10 7.63 8.23 8.17 8.17 8.47 8.10 8.40
Average Response times
8.38 5.73 3.65 2.90 2.62 2.06 1.79 1.63 1.63 1.38 1.24 1.18 1.03 1.20 0.98 0.99 1.01 0.89 0.92 0.89
```

The response times have gotten smaller, but the scores have dropped by about a point as well. To see why that is happening you would have to look at the history of production usage and utilities that are learned, which we will not do here.

That brings up the final issue that we will discuss. Adjusting the parameters for a model which uses production compilation can be a more difficult process than for other models. That is because of the potential for indirect effects to occur because of the automatic composition of new productions. Thus, unlike other models where the parameters often map fairly directly onto behavior, now one also has to consider what new productions can be learned and how the parameters affect those as well. Those effects may not always be in the same direction as one would expect (for example a faster production compilation learning rate leading to fewer correct responses). So, just like the extra work that was required to test the model to make sure it operated correctly, adjusting the parameters can also require looking at the new productions which are created and how their utilities are changing as a result of parameter adjustments when trying to achieve a particular fit to data or other explicit result from the model.